P&S Heterogeneous Systems

Dynamic Parallelism

Dr. Juan Gómez Luna
Prof. Onur Mutlu
ETH Zürich

Fall 2022

9 January 2023

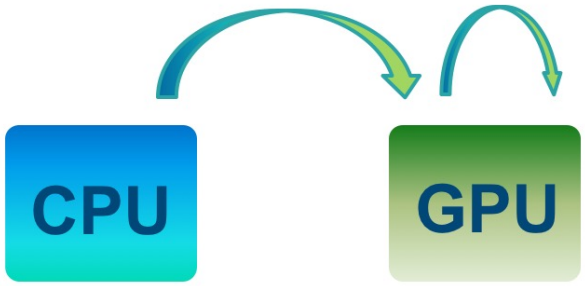
Dynamic Parallelism

- GPU programming frameworks provide an interface to express dynamic refinement algorithms in a more natural way
  - This dynamic parallelism interface allows GPU threads to launch GPU kernels when new work is dynamically discovered
  - Recall BFS
    - Each node in the frontier has a different number of neighbors
- CUDA Dynamic Parallelism
  - Important semantics when a kernel is launched from a kernel
  - Performance considerations

Dynamic Parallelism in CUDA

- Device-side kernel launches
  - Kepler GK110 architecture
  - Typical use cases
    - Dynamic load balancing
    - Data-dependent execution
    - Recursion
    - Library (with kernels) calls from kernels
  - Programmability and maintainability

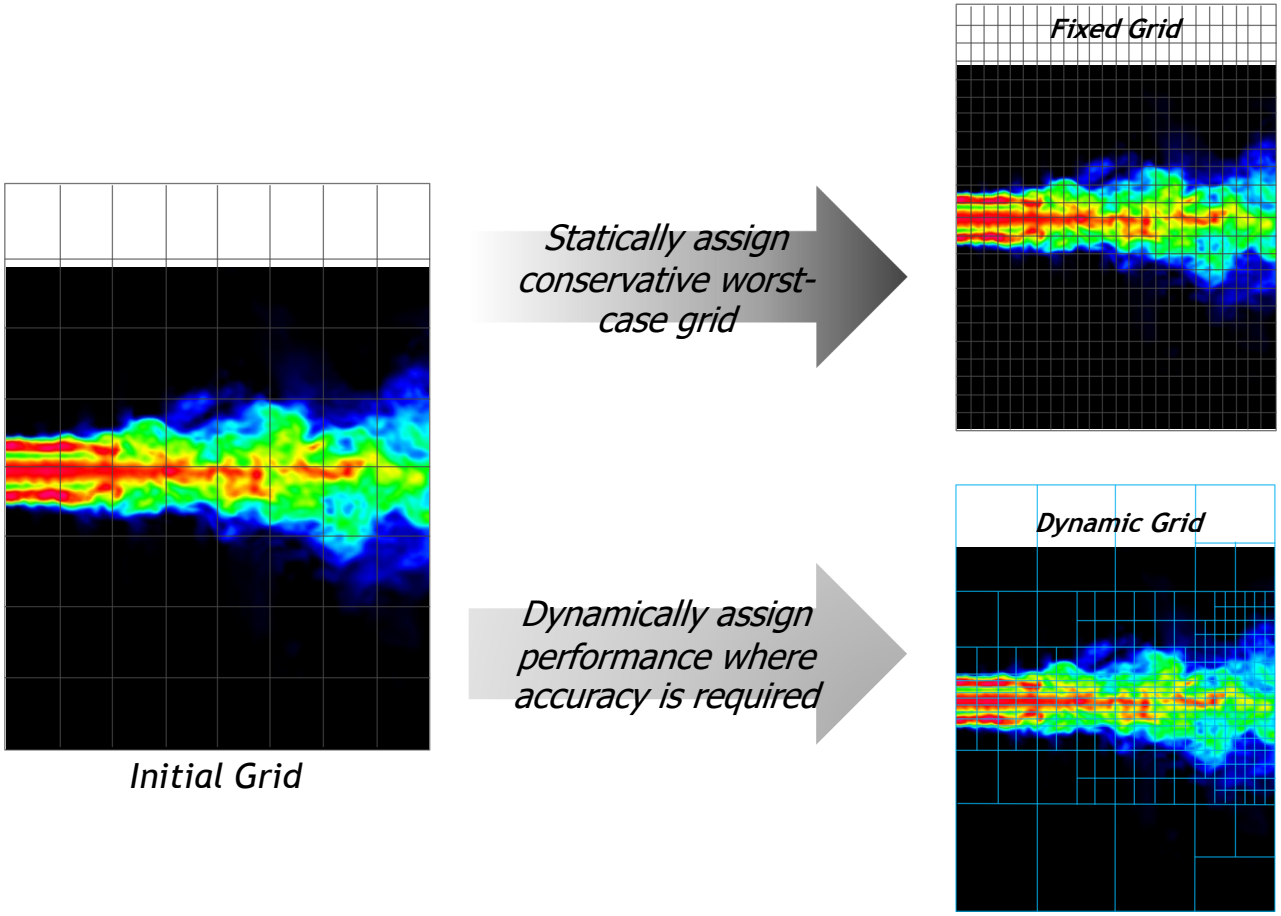
Fermi: Only CPU
can generate GPU work.



Kepler: GPU can generate work for itself.

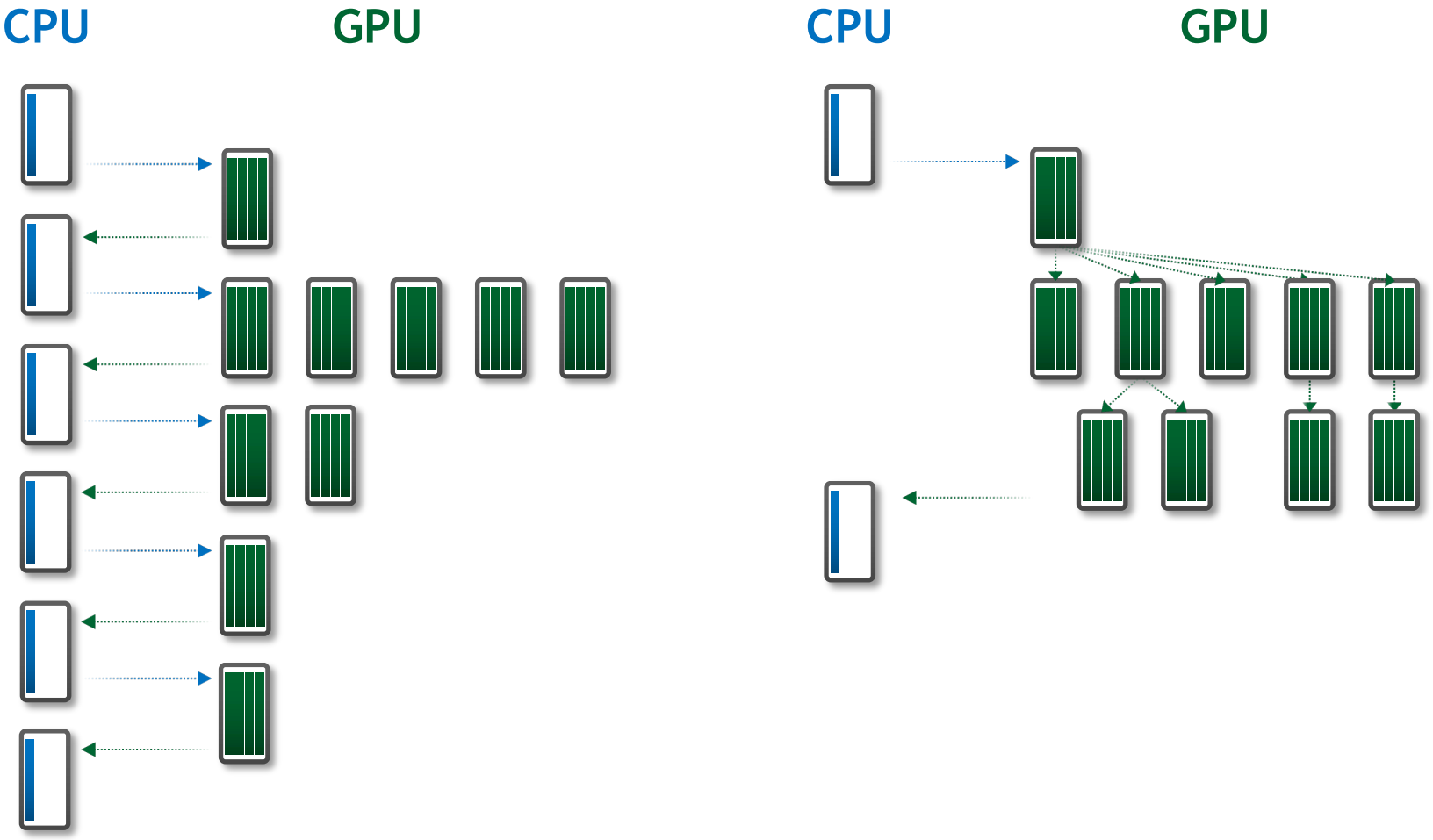
Example: Turbulence Simulation

- Fixed grid vs. dynamic grid for a turbulence simulation model



Kernel Launch with/without DP

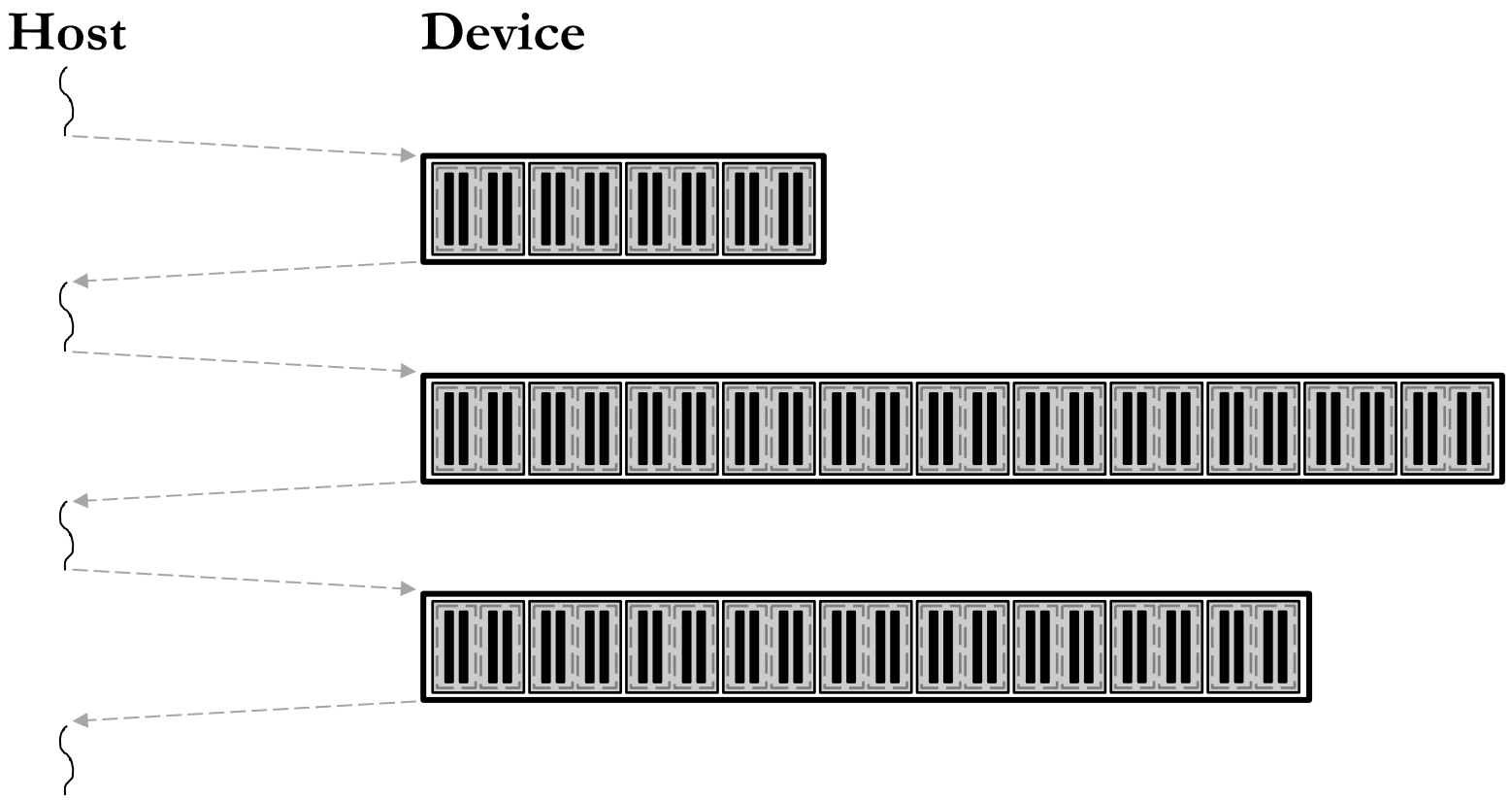
- CPU-GPU without and with dynamic parallelism



(a) Without Dynamic Parallelism

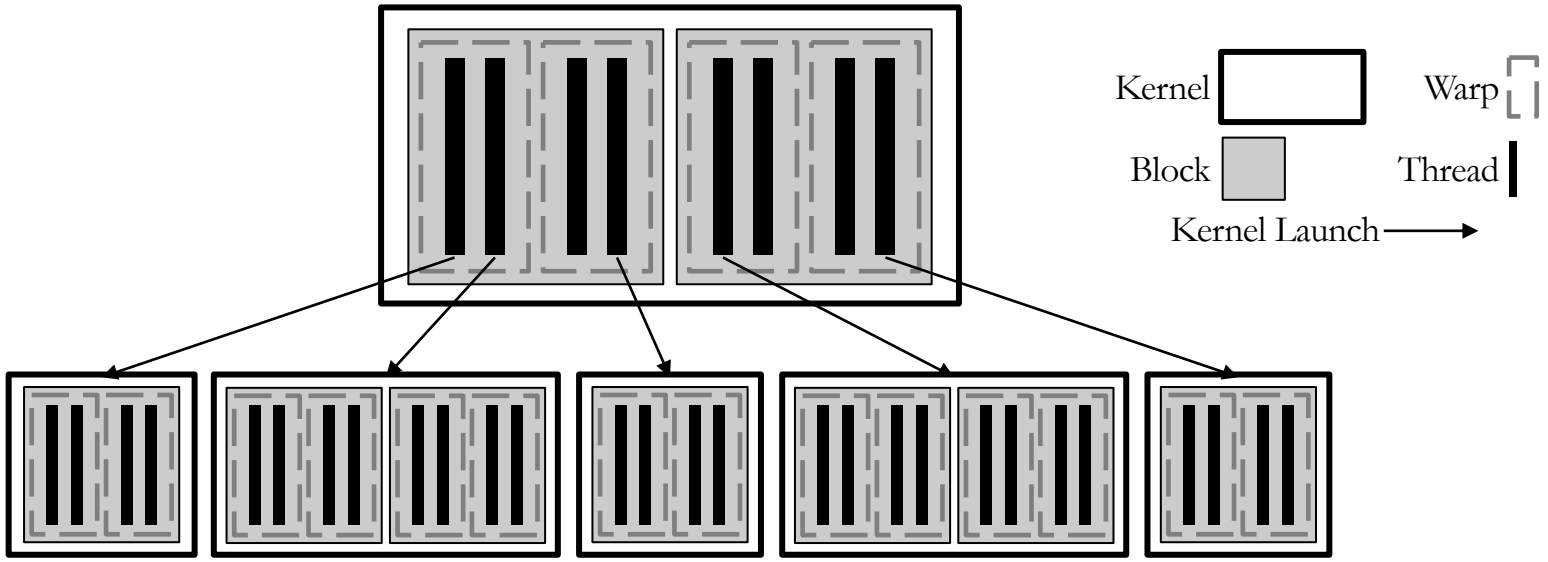
(b) With Dynamic Parallelism

Kernel Launch without Dynamic Parallelism



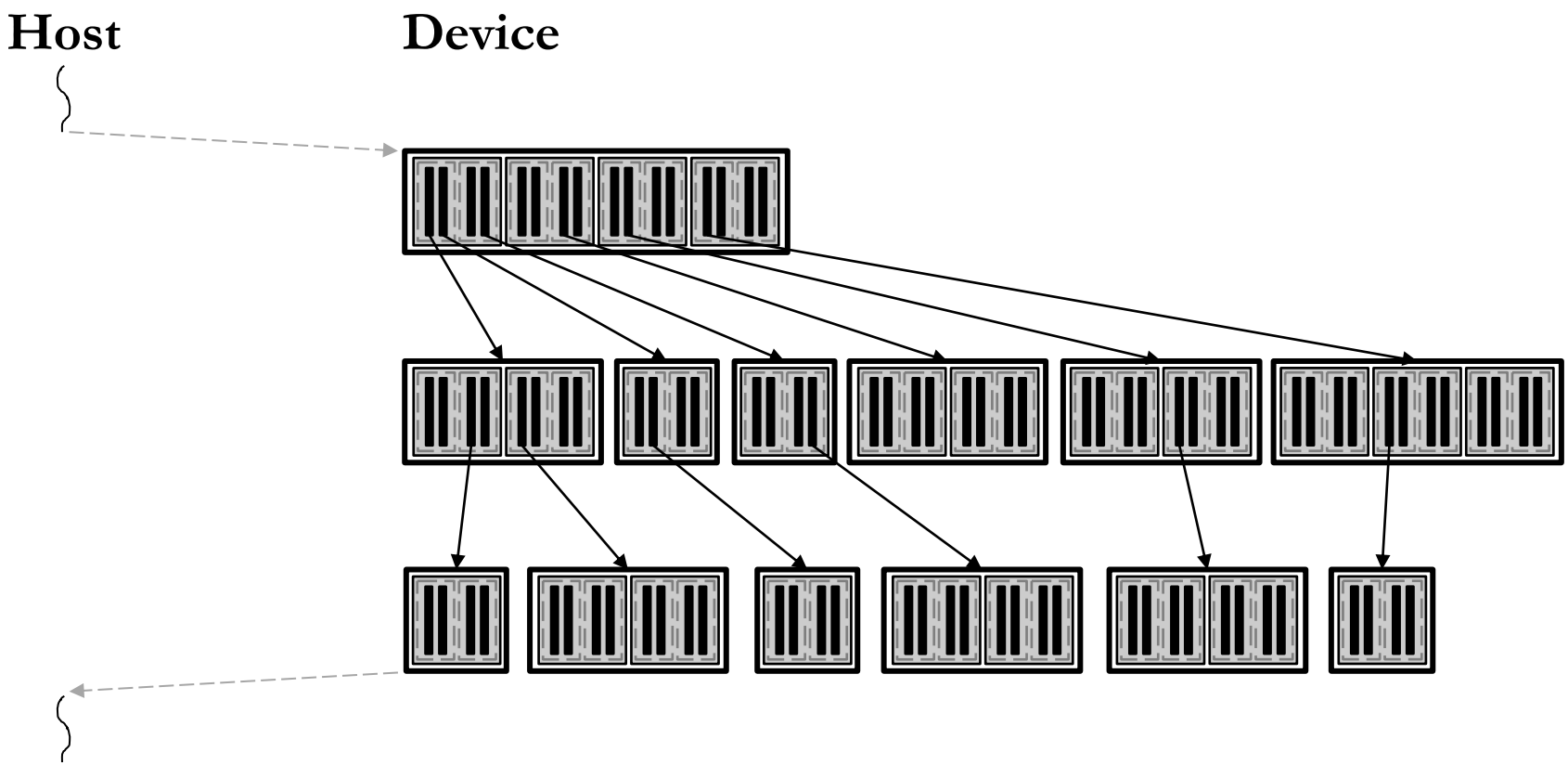
Previously, kernels could only be launched from the host (painful to program!)

Dynamic Parallelism



Kernels threads can launch new kernels on the device without host communication

Kernel Launch with Dynamic Parallelism



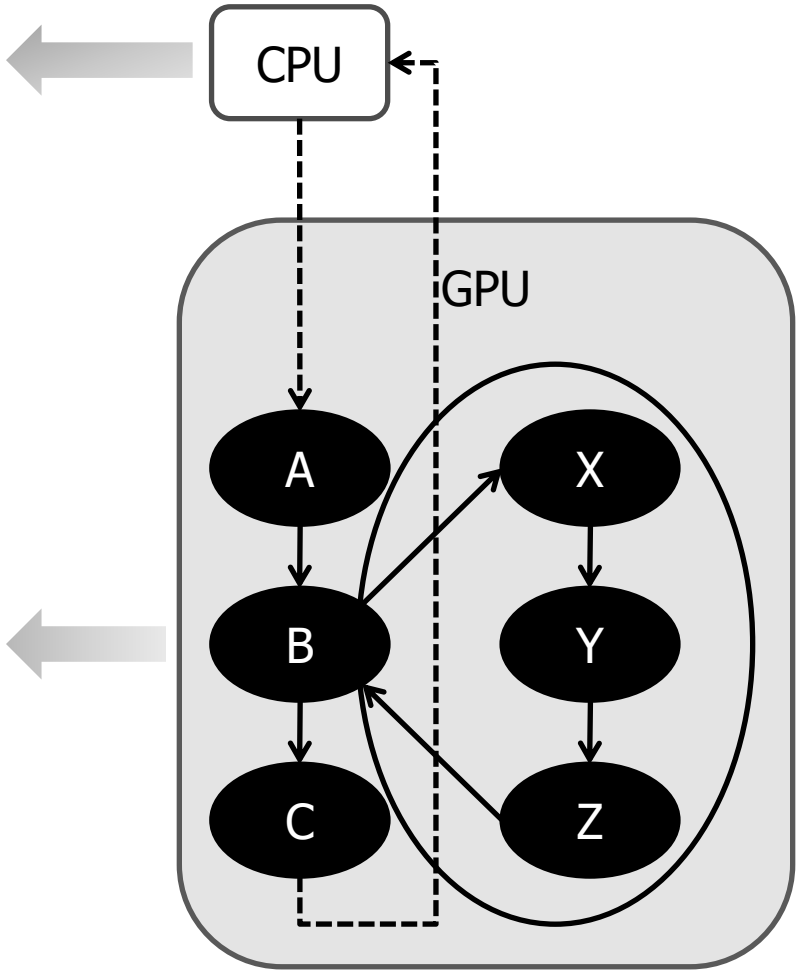
Easier to write programs with dynamically discovered parallelism

Nested Dependencies

- `B` is the parent kernel of `X`, `Y`, and `Z`
- `X`, `Y`, and `Z` are child kernels of `B`

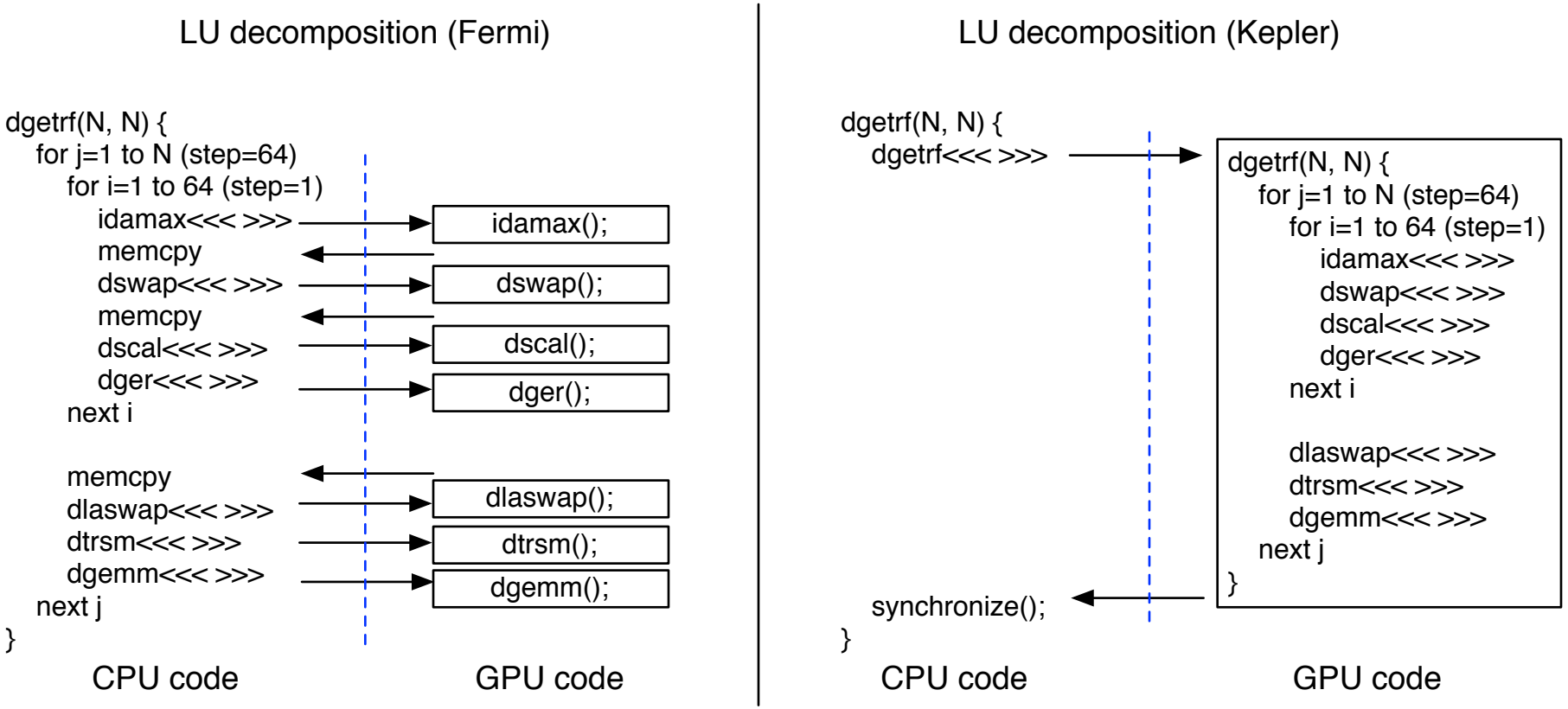
```
int main() {
    float *data;
    setup(data);

    A <<< ... >>> (data);
    B <<< ... >>> (data);
    C <<< ... >>> (data);

    cudaDeviceSynchronize();

    return 0;
}
```

```
__global__ void B(float *data)
{
    do_stuff(data);

     X <<< ... >>> (data);
     Y <<< ... >>> (data);
     Z <<< ... >>> (data);

     cudaDeviceSynchronize();

     do_more_stuff(data);
}
```



Example: LU Decomposition

- Solving systems of linear equations



Syntax for Child Kernel Launch

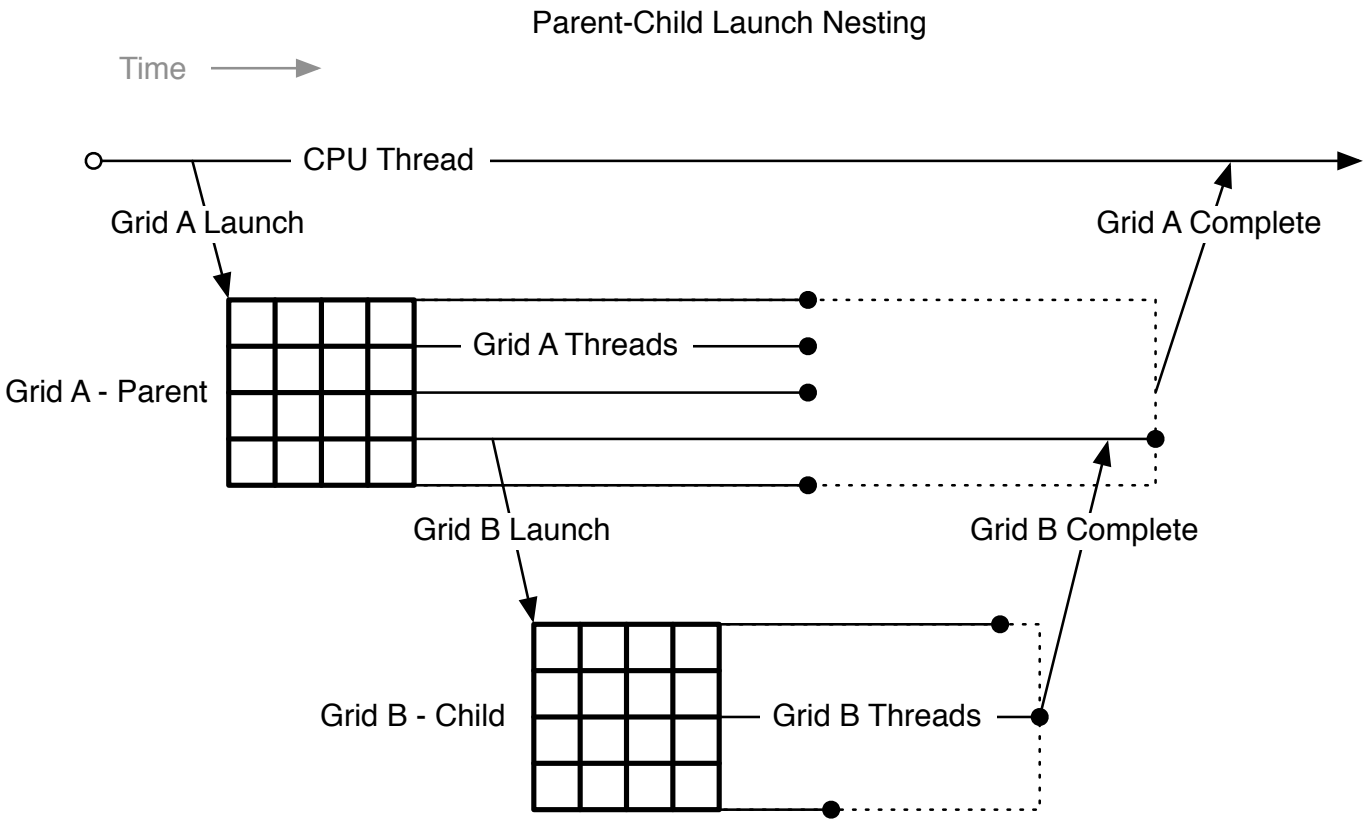
- A parent kernel launches a child kernel with the same syntax as the host

```
kernel_name<<< Dg, Db, Ns, S >>>([kernel arguments]);
```

  - `Dg` is of type `dim3` and specifies the dimensions and size of the grid
  - `Db` is of type `dim3` and specifies the dimensions and size of each thread block
  - `Ns` is of type `size_t` and specifies the number of bytes of shared memory that is dynamically allocated per thread block for this call
  - `S` is of type `cudaStream_t` and specifies the stream associated with this call

Parent-Child Synchronization

- Synchronization
  - Parent to child: memory consistency
  - Child to parent: after `cudaDeviceSynchronize()`

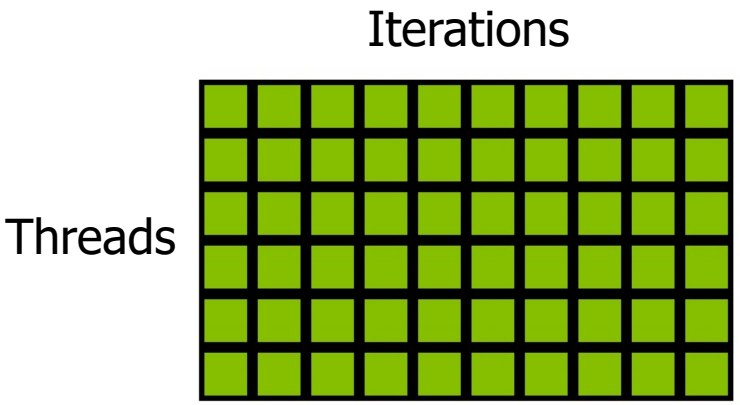


A Simple Example

A Simple Example (I)

- Without dynamic parallelism

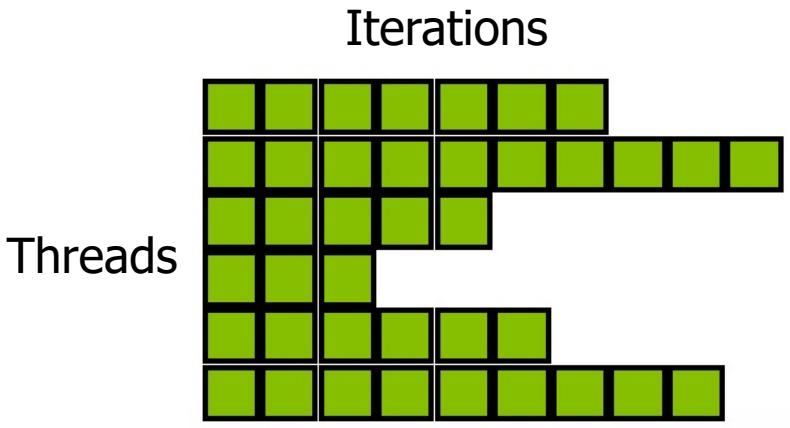
```
01    __global__ void kernel(unsigned int start, unsigned int end,
02        float* someData, float* moreData) {
03
04        unsigned int i = blockIdx.x*blockDim.x + threadIdx.x;
05        doSomeWork(someData[i]);
06
07        for(unsigned int j = start; j < end; ++j) {
08            doMoreWork(moreData[j], i);
09        }
10
11    }
```



A Simple Example (II)

- Without dynamic parallelism, non-uniform workload

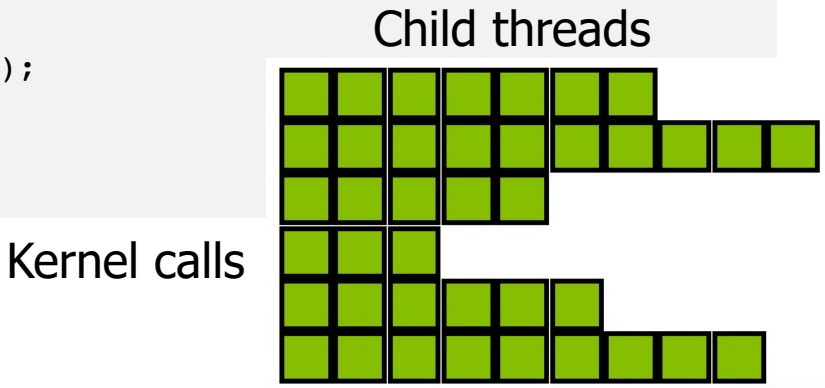
```
01    __global__ void kernel(unsigned int* start, unsigned int* end,
02        float* someData, float* moreData) {
03
04        unsigned int i = blockIdx.x*blockDim.x + threadIdx.x;
05        doSomeWork(someData[i]);
06
07        for(unsigned int j = start[i]; j < end[i]; ++j) {
08            doMoreWork(moreData[j]);
09        }
10
11    }
```



A Simple Example (III)

- With dynamic parallelism, non-uniform workload

```
01    __global__ void kernel_parent(unsigned int* start, unsigned int* end,
02        float* someData, float* moreData) {
03
04        unsigned int i = blockIdx.x*blockDim.x + threadIdx.x;
05        doSomeWork(someData[i]);
06
07        kernel_child <<< ceil((end[i]-start[i])/256.0) , 256 >>>
08            (start[i], end[i], moreData);
09
10    }
11
12    __global__ void kernel_child(unsigned int start, unsigned int end,
13        float* moreData) {
14
15        unsigned int j = start + blockIdx.x*blockDim.x + threadIdx.x;
16
17        if(j < end) {
18            doMoreWork(moreData[j]);
19        }
20
21    }
```



A More Complex Example

Bezier Lines (I)

- Linear Bezier curves

$$\mathbf{B}(t) = \mathbf{P}_0 + t(\mathbf{P}_1 - \mathbf{P}_0) = (1-t)\mathbf{P}_0 + t\mathbf{P}_1, t \in [0,1]$$

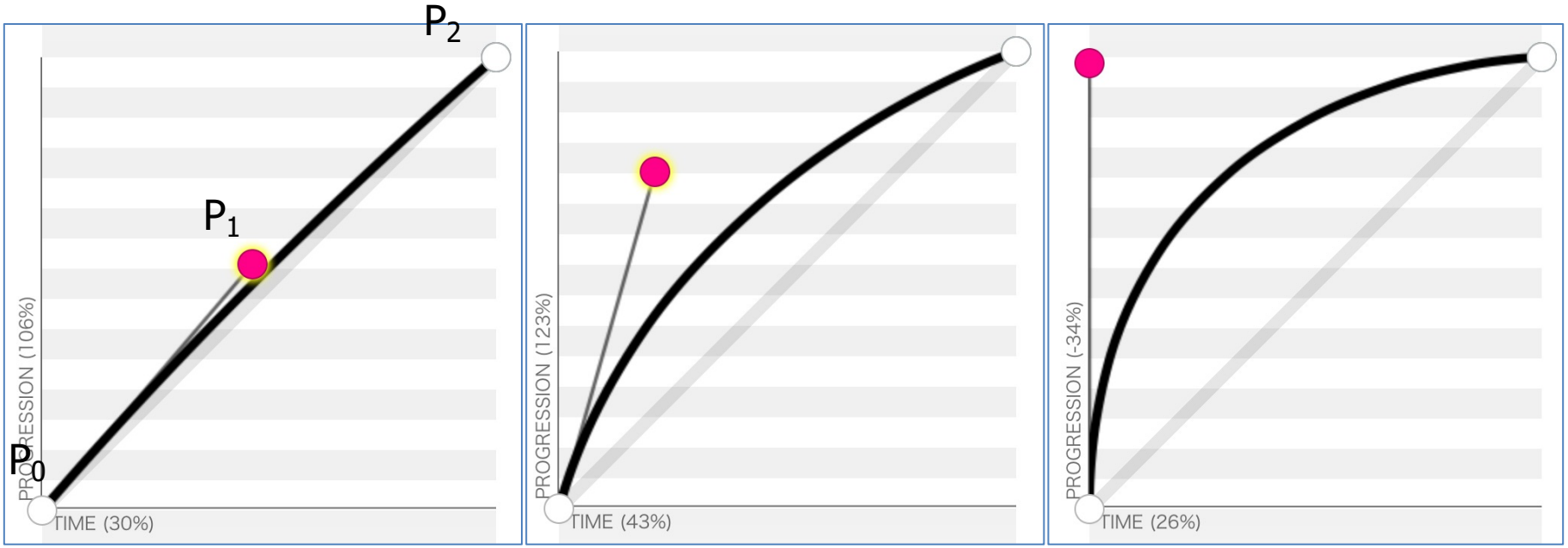
- Quadratic Bezier curves

$$\mathbf{B}(t) = (1-t)[(1-t)\mathbf{P}_0 + t\mathbf{P}_1] + t[(1-t)\mathbf{P}_0 + t\mathbf{P}_2], t \in [0,1],$$

$$\mathbf{B}(t) = (1-t)^2\mathbf{P}_0 + 2(1-t)t\mathbf{P}_1 + t^2\mathbf{P}_2, t \in [0,1]$$

Bezier Lines (II)

- Control points
  - Curvature calculation
  - Tessellation points



http://cubic-bezier.com

Bezier Lines without Dynamic Parallelism

- Without dynamic parallelism: One line per block

```
046   __global__ void computeBezierLines(BezierLine *bLines, int nLines) {
047     int bidx = blockIdx.x;
048     if(bidx < nLines){
049       //Compute the curvature of the line
050       float curvature = computeCurvature(bLines);
051
052       //From the curvature, compute the number of tessellation points
053       int nTessPoints = min(max((int)(curvature*16.0f),4),32);
054       bLines[bidx].nVertices = nTessPoints;
055
056       //Loop through vertices to be tessellated, incrementing by blockDim.x
057       for(int inc = 0; inc < nTessPoints; inc += blockDim.x){
058         int idx = inc + threadIdx.x;  //Compute a unique index for this point
059         if(idx < nTessPoints){
060           float u = (float)idx/(float)(nTessPoints-1);  //Compute u from idx
061           float omu = 1.0f - u;   //pre-compute one minus u
062
063           float B3u[3]; //Compute quadratic Bezier coefficients
064           B3u[0] = omu*omu;
065           B3u[1] = 2.0f*u*omu;
066           B3u[2] = u*u;
067
068           float2 position = {0,0};  //Set position to zero
069           for(int i = 0; i < 3; i++){
070             //Add the contribution of the i'th control point to position
071             position = position + B3u[i] * bLines[bidx].CP[i];
072           }
073           //Assign value of vertex position to the correct array element
074           bLines[bidx].vertexPos[idx] = position;
075         }
076       }
077   }
```

Bezier Lines with Dynamic Parallelism (I)

- With dynamic parallelism
- Parent: One line per thread

```
30  __global__ void computeBezierLines_parent(BezierLine *bLines, int nLines) {
31    //Compute a unique index for each Bezier line
32    int lidx = threadIdx.x + blockDim.x*blockIdx.x;
33    if(lidx < nLines){
34      //Compute the curvature of the line
35      float curvature = computeCurvature(bLines);
36      //From the curvature, compute the number of tessellation points
37      bLines[lidx].nVertices = min(max((int)(curvature*16.0f),4),MAX_TESS_POINTS);
38      cudaMalloc((void**)&bLines[lidx].vertexPos, bLines[lidx].nVertices*sizeof(float2));
39      //Call the child kernel to compute the tessellated points for each line
40      computeBezierLine_child<<<ceil((float)bLines[lidx].nVertices/32.0f), 32>>>
41        (lidx, bLines, bLines[lidx].nVertices);
42    }
43  }
```

Bezier Lines with Dynamic Parallelism (II)

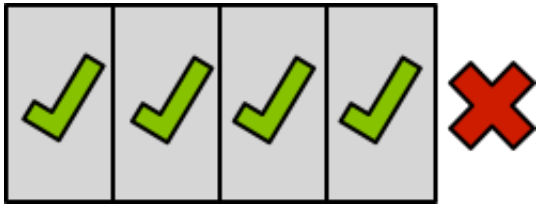
- With dynamic parallelism
- Child

```
07    __global__ void computeBezierLine_child(int lidx, BezierLine* bLines,
08      int nTessPoints) {
09      int idx = threadIdx.x + blockDim.x*blockIdx.x;  //Compute idx unique to this vertex
10      if(idx < nTessPoints){
11        float u = (float)idx/(float)(nTessPoints-1);  //Compute u from idx
12        float omu = 1.0f - u;   //Pre-compute one minus u
13
14        float B3u[3];   //Compute quadratic Bezier coefficients
15        B3u[0] = omu*omu;
16        B3u[1] = 2.0f*u*omu;
17        B3u[2] = u*u;
18
19        float2 position = {0,0};  //Set position to zero
20        for(int i = 0; i < 3; i++) {
21          //Add the contribution of the i'th control point to position
22          position = position + B3u[i] * bLines[lidx].CP[i];
23        }
24
25        //Assign the value of the vertex position to the correct array element
26        bLines[lidx].vertexPos[idx] = position;
27      }
28    }
```

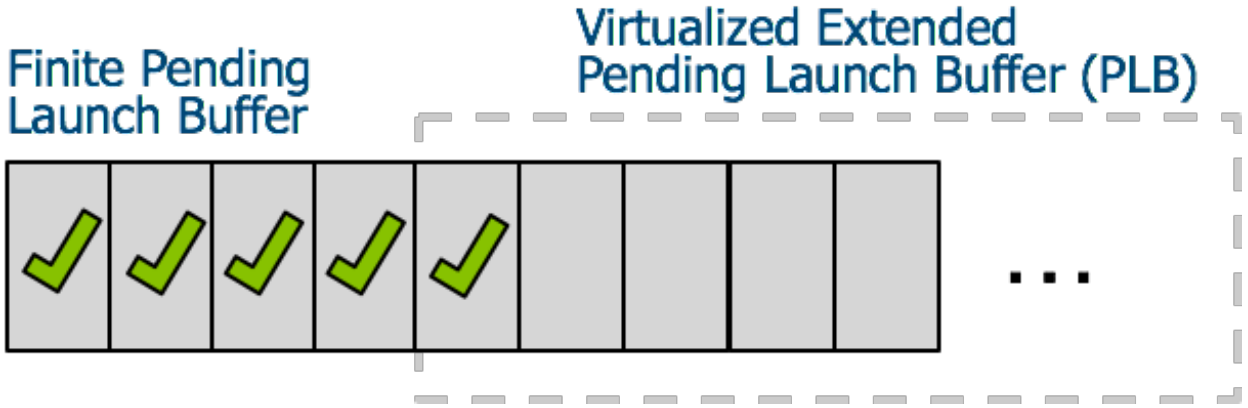
Launch Pool (I)

- Launch pool size
  - Fixed-size pool: default 2048
  - Variable-size pool

Before CUDA 6.0

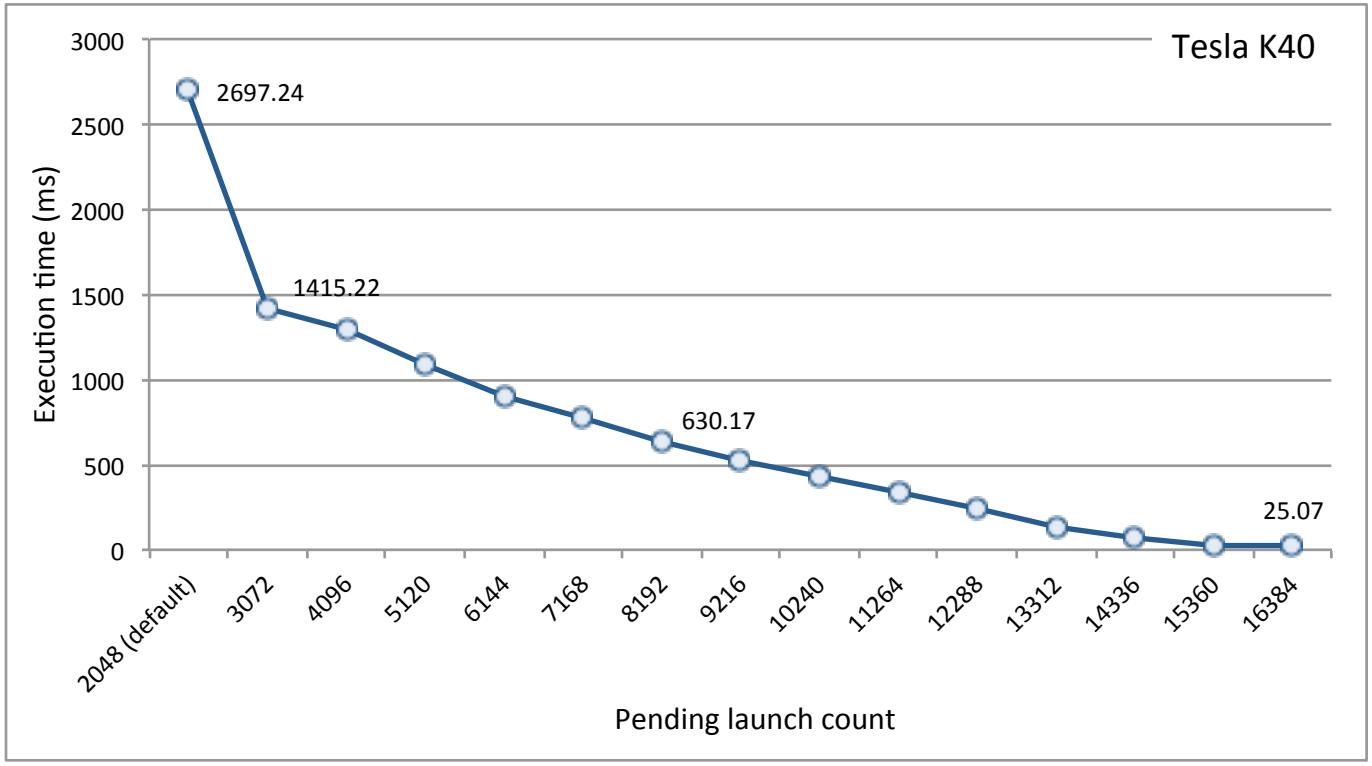


Since CUDA 6.0



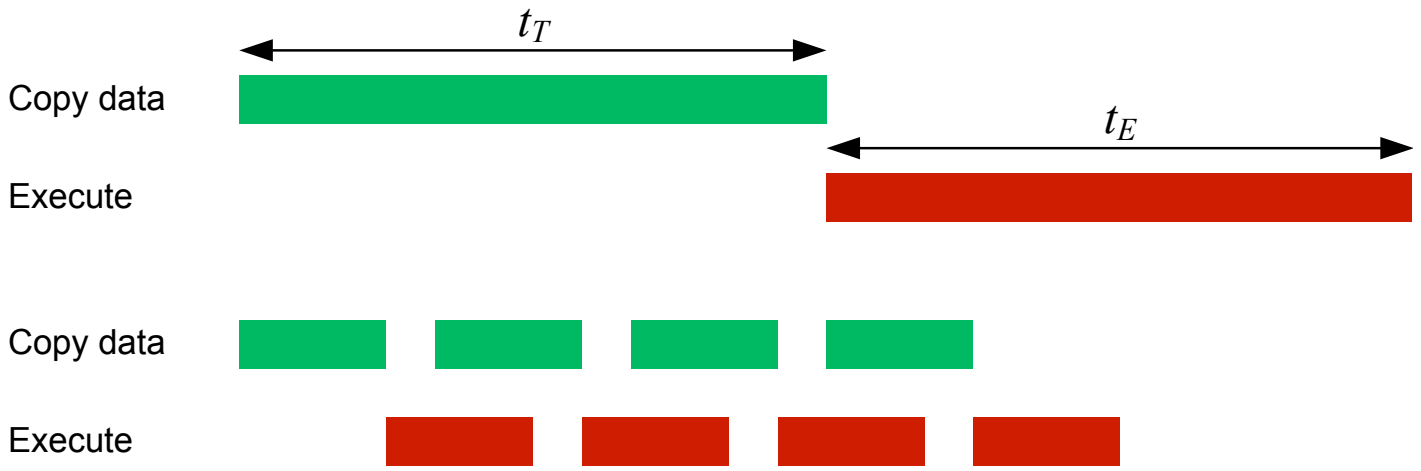
Launch Pool (II)

- Effect of increasing the fixed-size launch pool
  - Number of Bezier lines = 16384



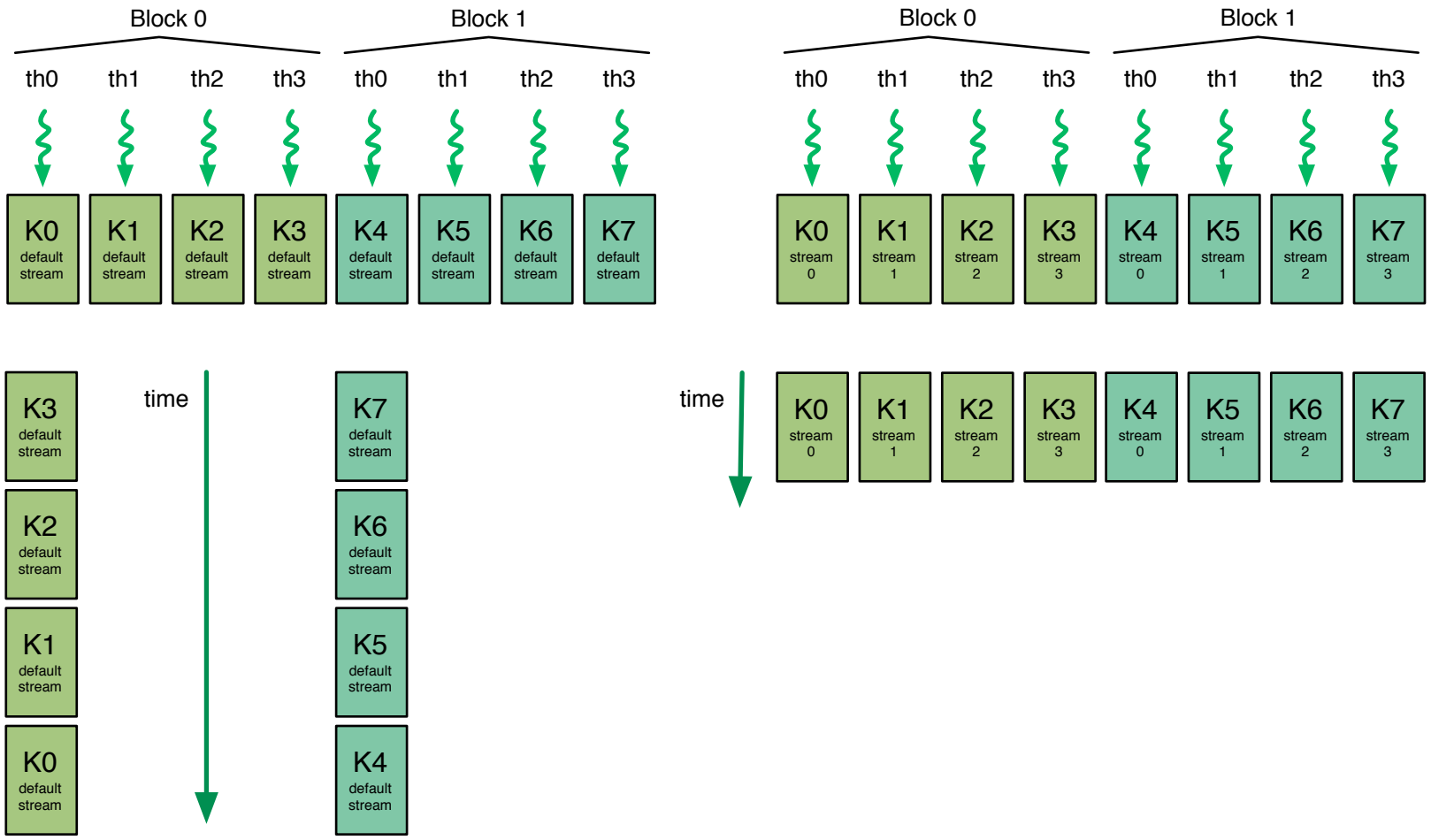
CUDA Streams

- CUDA streams (command queues in OpenCL)
- Sequence of operations that are performed in order
  - 1. Data transfer CPU-GPU
  - 2. Kernel execution
    - D input data instances, B blocks
    - #Streams: (D / #Streams) data instances, (B / #Streams) blocks
  - 3. Data transfer GPU-CPU



Streams for Child Kernel Launch (I)

- Default stream vs. several streams per parent block



Streams for Child Kernel Launch (II)

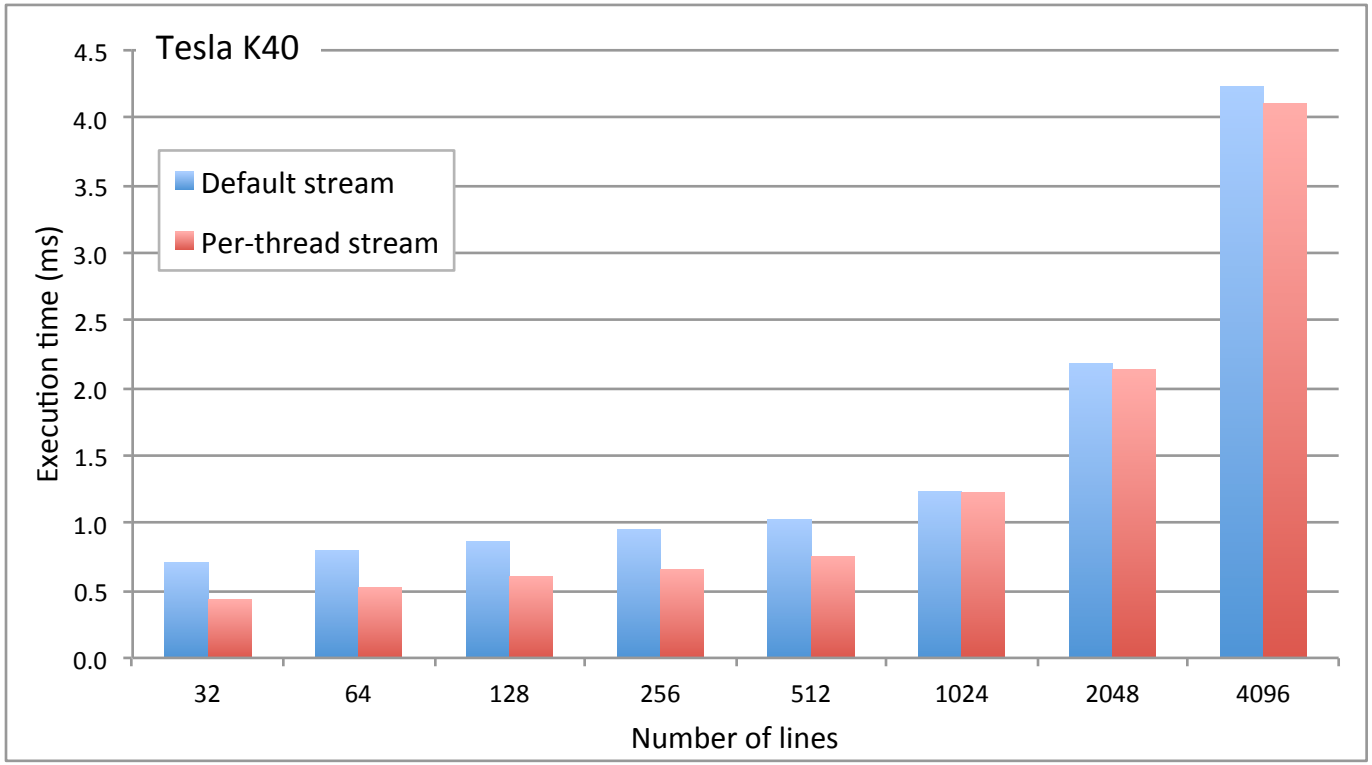
- Default stream vs. several streams per parent block

```
// Call the child kernel to compute the tessellated points for each line
// Default stream
computeBezierLine_child<<<ceil((float)bLines[lidx].nVertices/32.0f), 32>>>
    (lidx, bLines, bLines[lidx].nVertices);
```

```
cudaStream_t stream;
// Create non-blocking stream
cudaStreamCreateWithFlags(&stream, cudaStreamNonBlocking);

// Call the child kernel to compute the tessellated points for each line
computeBezierLine_child<<<ceil((float)bLines[lidx].nVertices/32.0f), 32, 0, stream>>>
    (lidx, bLines, bLines[lidx].nVertices);

// Destroy stream
cudaStreamDestroy(stream);
```

Streams for Child Kernel Launch (III)

- Performance impact of per-thread streams



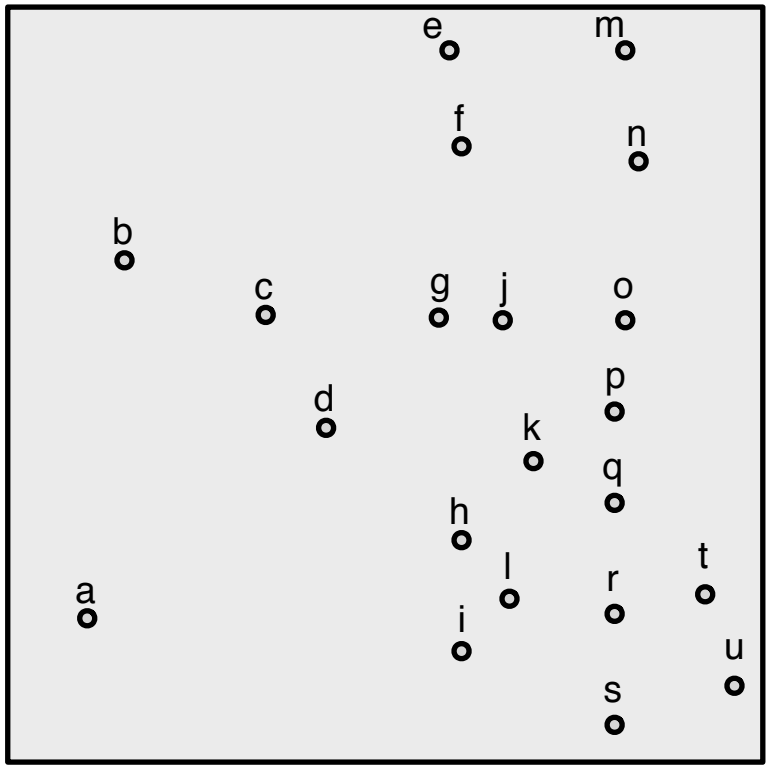
A Recursive Example

Arranging Input for Gather Parallelization

- Input data distribution in space may be far from uniform
  - Quad/Oct Trees
- One may not be able to pre-determine the input elements needed for calculating output elements
  - Dynamic Search
- …

A Recursive Example: Quadtree (I)

- Partitioning a 2D space by recursively dividing it into four quadrants until the number of atoms in each quadrant is less than a threshold

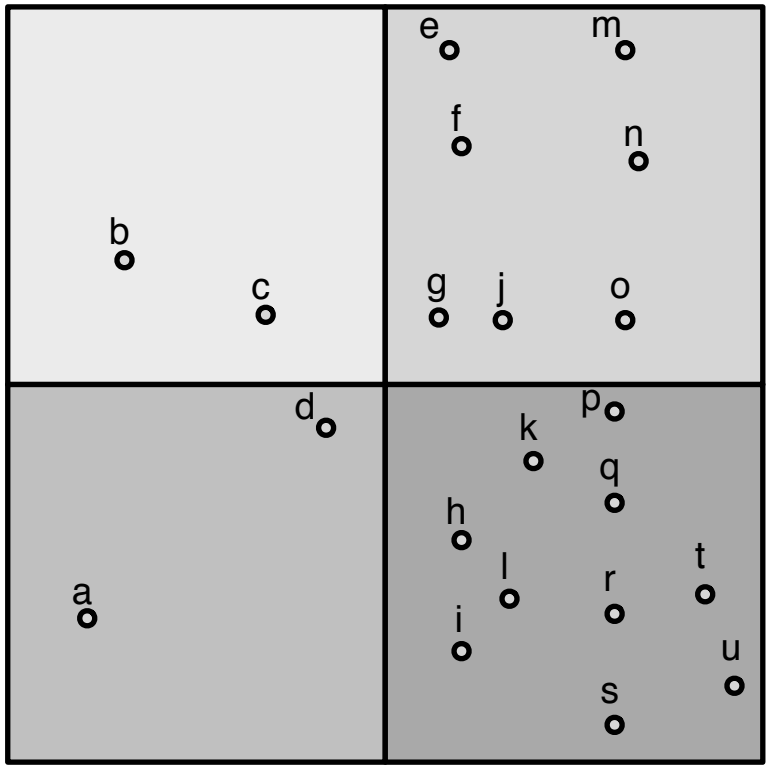


Depth = 0

Threshold = 2

A Recursive Example: Quadtree (II)

- Partitioning a 2D space by recursively dividing it into four quadrants until the number of atoms in each quadrant is less than a threshold

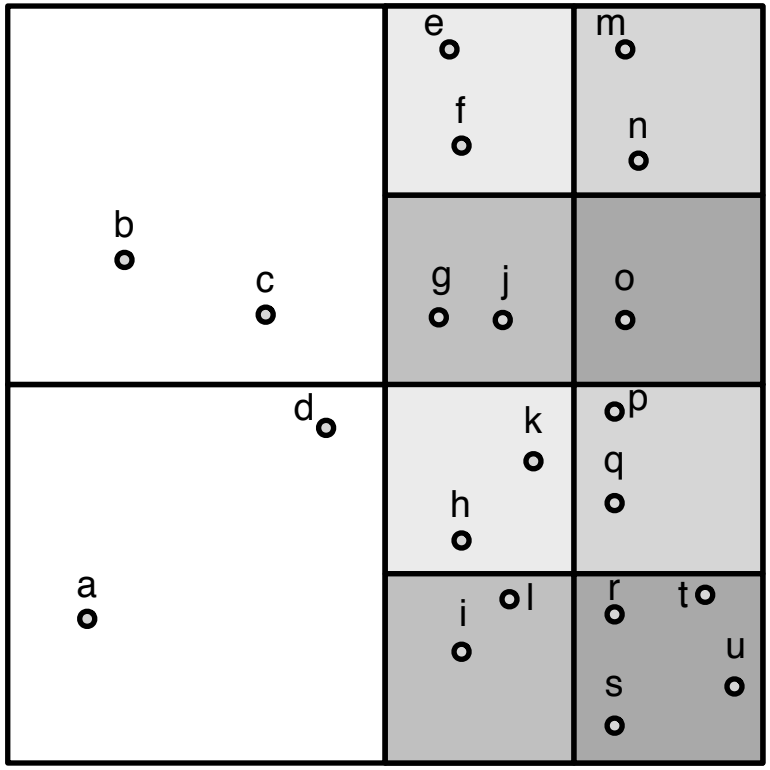


Depth = 1

Threshold = 2

A Recursive Example: Quadtree (III)

- Partitioning a 2D space by recursively dividing it into four quadrants until the number of atoms in each quadrant is less than a threshold

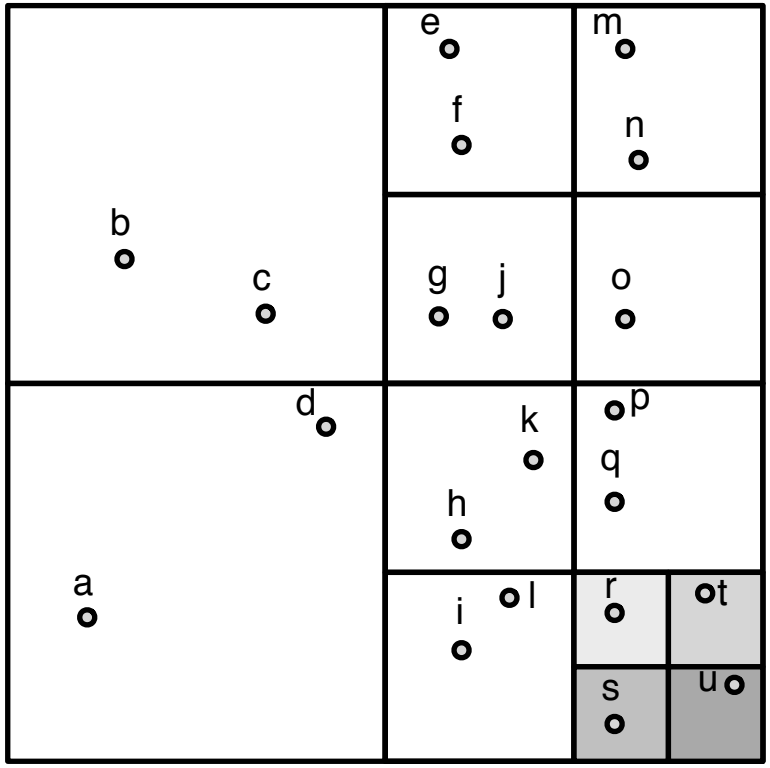


Depth = 2

Threshold = 2

A Recursive Example: Quadtree (IV)

- Partitioning a 2D space by recursively dividing it into four quadrants until the number of atoms in each quadrant is less than a threshold

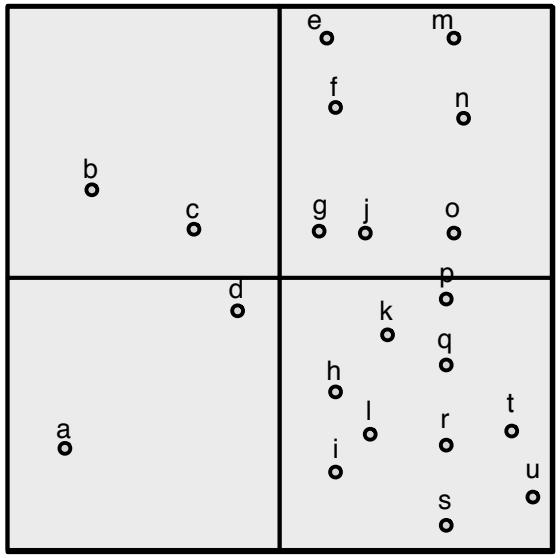


Depth = 3

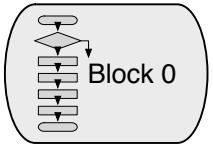
Threshold = 2

A Recursive Example: Quadtree (V)

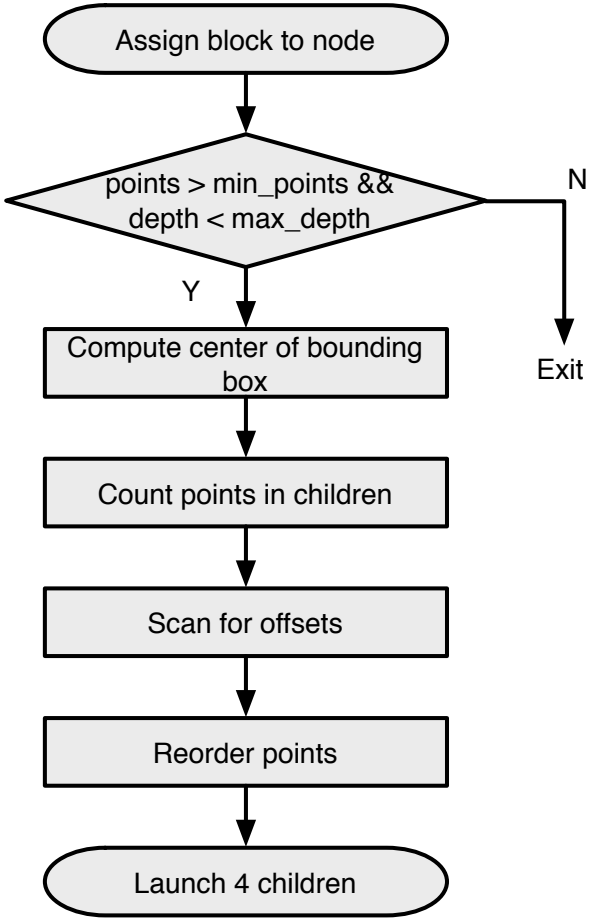
- 1 thread block is launched from host

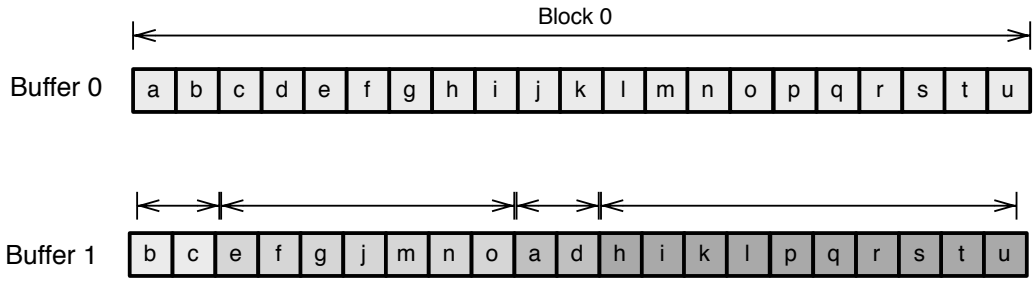


Depth = 0



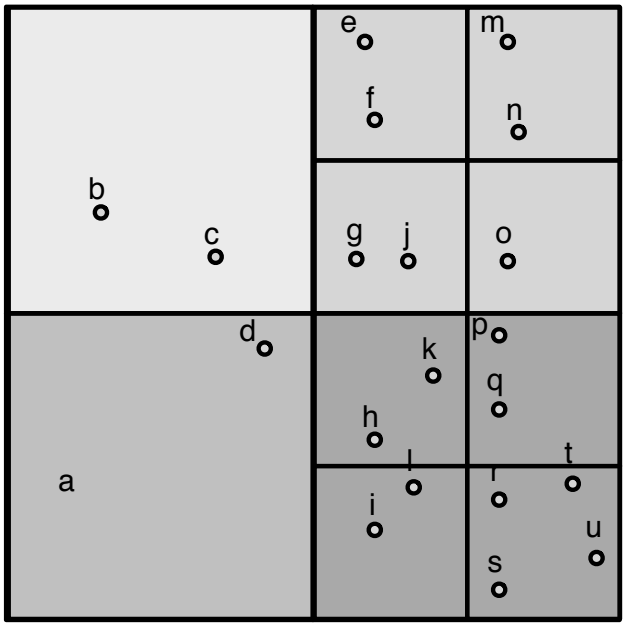
Outline of recursive kernel



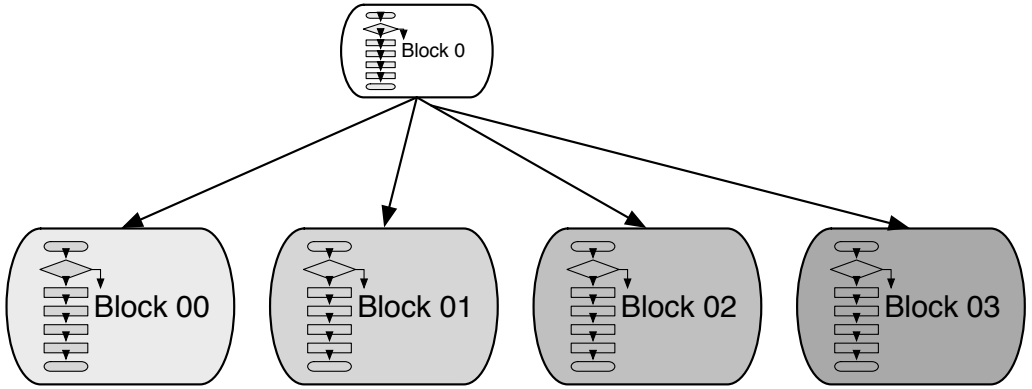


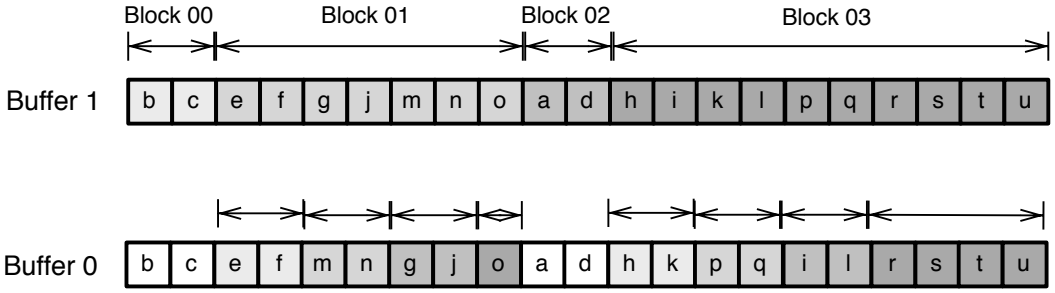
A Recursive Example: Quadtree (VI)

- Each block launches 1 child grid of 4 blocks



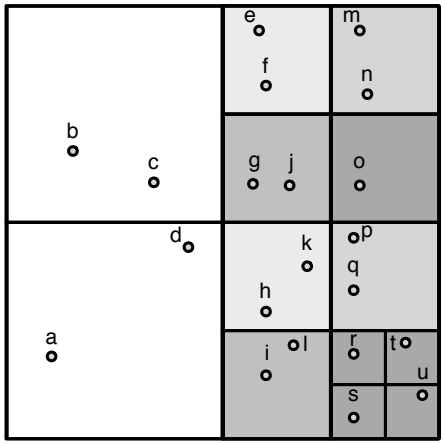
Depth = 1

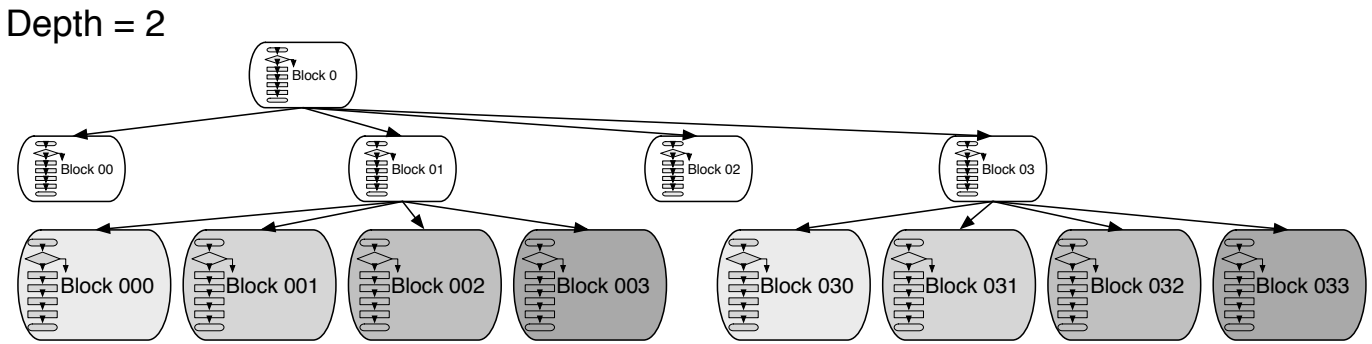


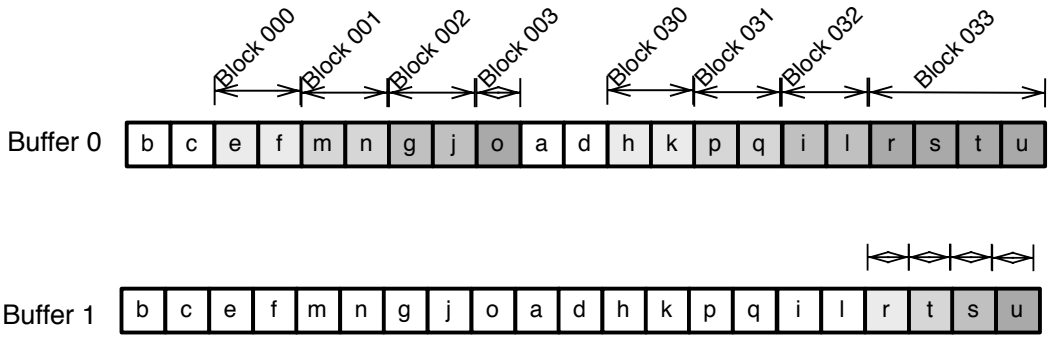


A Recursive Example: Quadtree (VII)

- Each block launches 1 child grid of 4 blocks

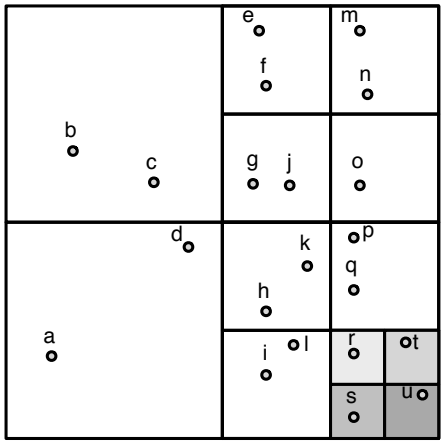


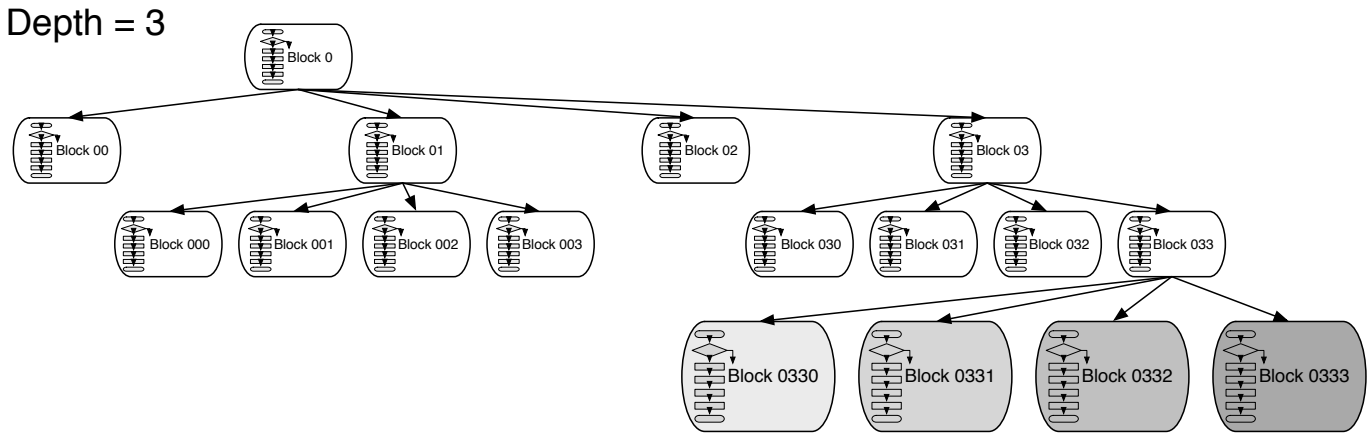


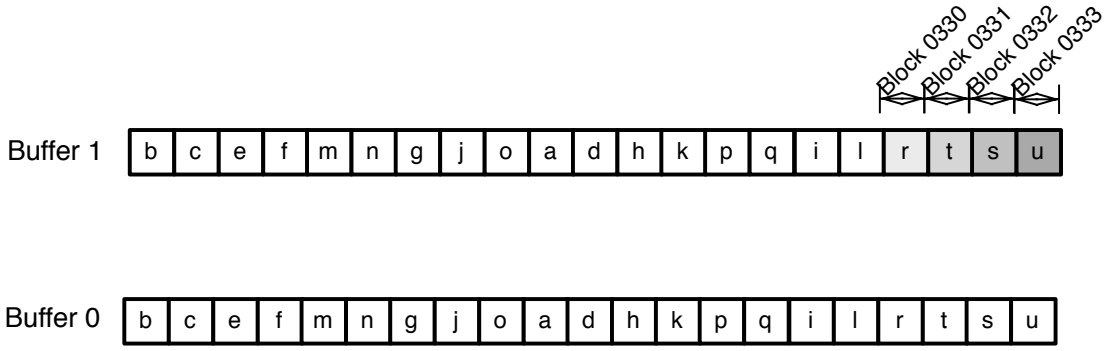


A Recursive Example: Quadtree (VIII)

- Each block launches 1 child grid of 4 blocks

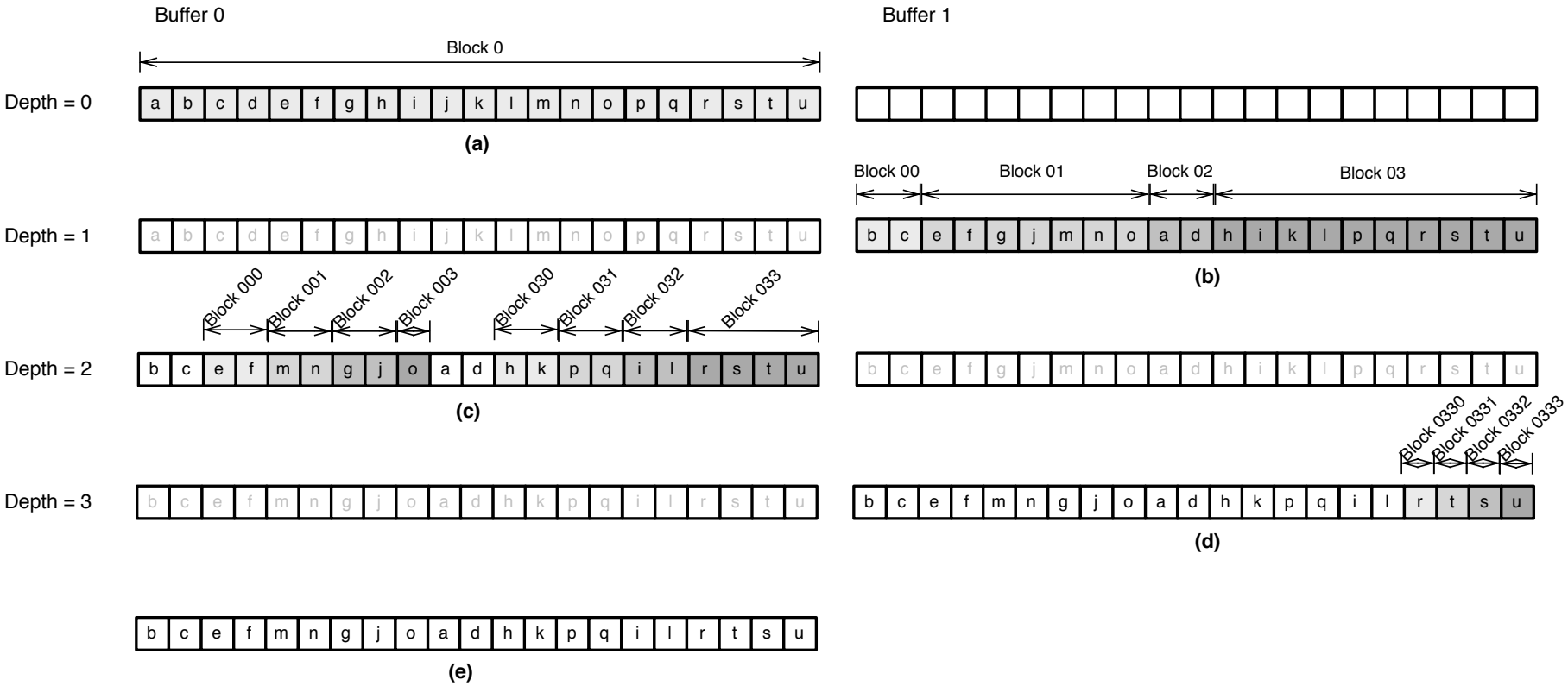






A Recursive Example: Quadtree (IX)

- Points in the same quadrant are grouped together



Each thread block may need to move its part to Buffer 0 before retiring, since there is where we expect the final output

Summary: Quadtree Construction

- The execution starts with host launching one thread block
  - At each recursion, if the number of atoms in the quadrant is less than or equal to the threshold, the thread block exits
- At each recursion, threads in each thread block that do not exit will collaboratively
  - Determine the number of atoms that belong in each quadrant
  - Perform a scan to determine the starting point of each quadrant
  - Reorder the atoms so that all atoms in the same quadrant are placed consecutively
  - One representative thread launches a kernel with 4 child blocks

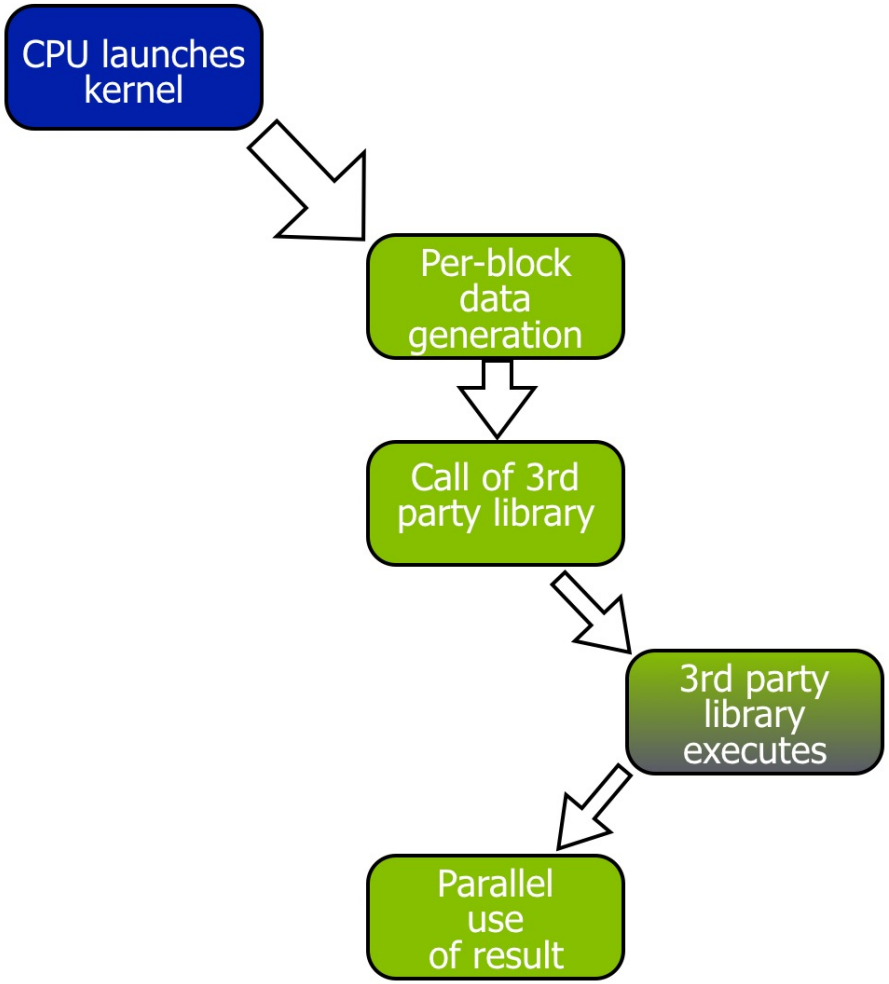
- Oct Tree is for 3D space
  - A 3D space is divided into 8 Octants
  - Each block that does not exit launches 1 child grid of 8 blocks

Other Use Cases

Use Case: Library Calls

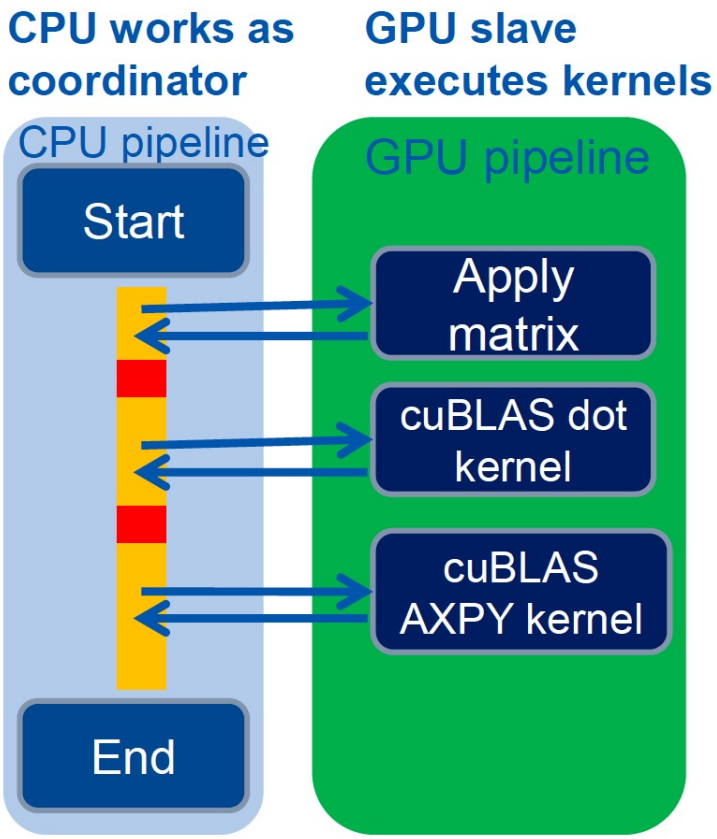
- Simple library calls

```
__global__ void libraryCall(float *a,
                            float *b,
                            float *c)
{
  // All threads generate data
  createData(a, b);
  __syncthreads();

  // The first thread calls library
  if (threadIdx.x == 0) {
    cublasDgemm(a, b, c);
    cudaDeviceSynchronize();
  }

  // All threads wait for results
  __syncthreads();

  consumeData(c);
}
```



Use Case: Avoid Launch Overhead (I)

- Lattice QCD



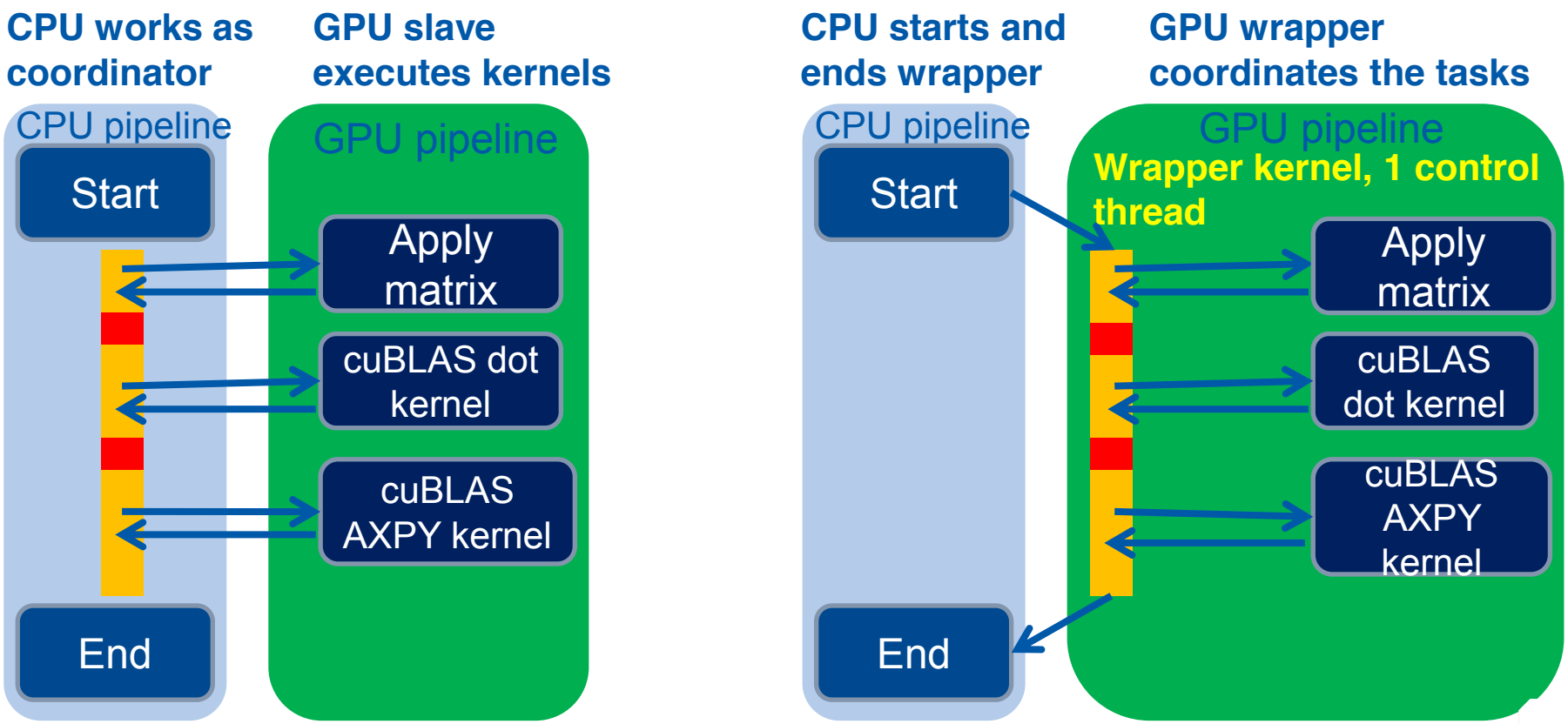
Bottlenecks

- Large number of calls to cuBLAS
- Dominated by CPU's capability of launching cuBLAS kernels
- ARM CPU is not fast enough to quickly launch kernels: GPU is underutilized

Vishal Mehta, "Exploiting CUDA Dynamic Parallelism for low power ARM based prototypes," GTC 2015

Use Case: Avoid Launch Overhead (II)

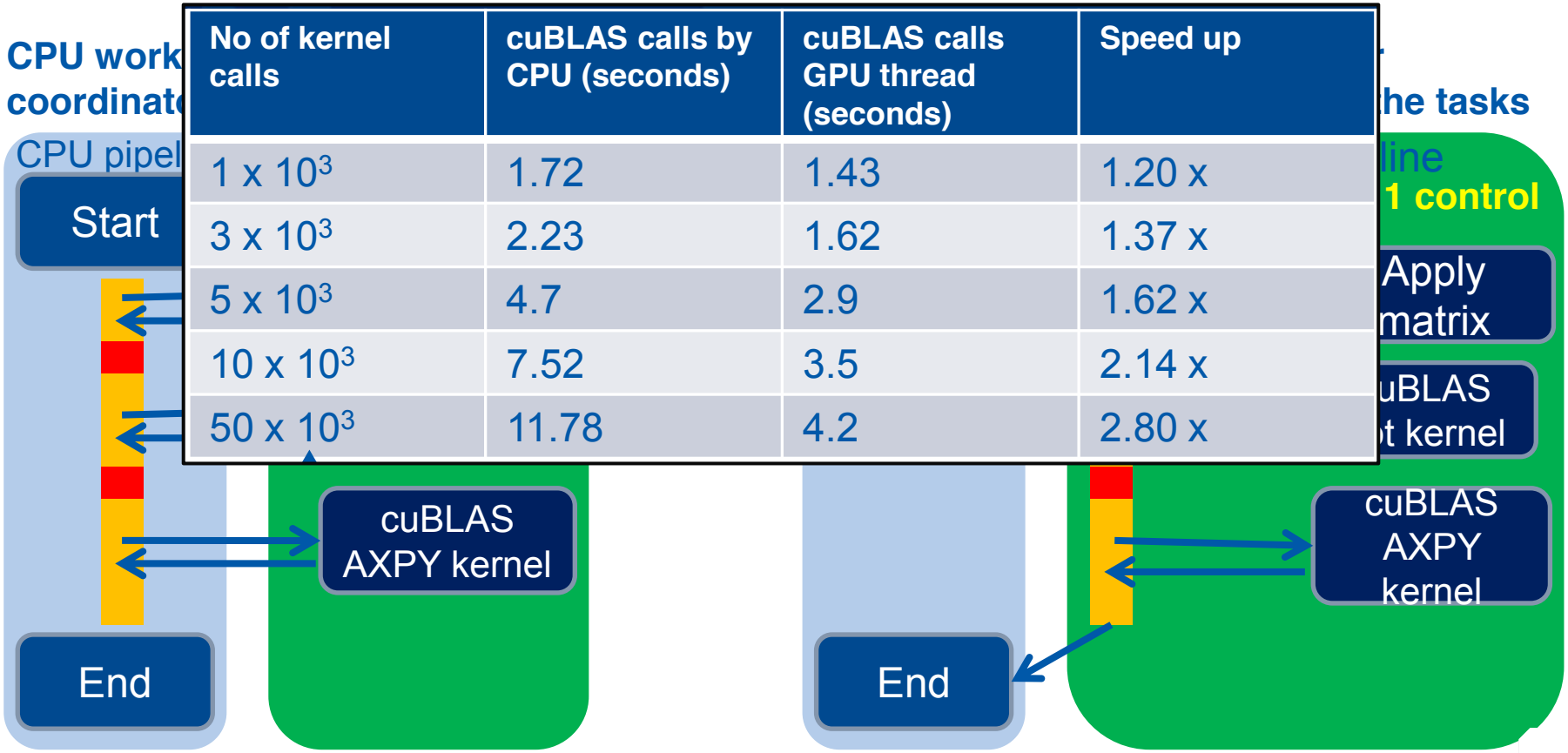
- Saving kernel launches in ARM CPUs



Use Case: Avoid Launch Overhead (III)

- Saving kernel launches in ARM CPUs

No of kernel calls	cuBLAS calls by CPU (seconds)	cuBLAS calls GPU thread (seconds)	Speed up
1 x 10^3	1.72	1.43	1.20 x
3 x 10^3	2.23	1.62	1.37 x
5 x 10^3	4.7	2.9	1.62 x
10 x 10^3	7.52	3.5	2.14 x
50 x 10^3	11.78	4.2	2.80 x



Performance Limitations

Performance Limitations

- Dynamic Parallelism ensures better work balance, and offers advantages in terms of programmability
- However, launching grids with a very small number of threads could lead to severe underutilization of the GPU resources
- A general recommendation
  - Child grids with a large number of thread blocks,
  - or at least thread blocks with hundreds of threads, if the number of blocks is small
- Nested parallelism (tree processing)
  - Thick tree nodes (each node deploys many threads),
  - and/or branch degree is large (each parent node has many children)
  - As the nesting depth is limited in hardware, only relatively shallow trees can be implemented efficiently

Optimization for Dynamic Parallelism

Alleviating Launch Overhead

- Dynamic Parallelism (CUDA, OpenCL2.0)
  - Dynamic load balancing
  - Data-dependent execution
  - Recursion
  - Programmability and maintainability

- Many fine-grain child kernels incur high kernel launch overhead and underutilization of the GPU resources

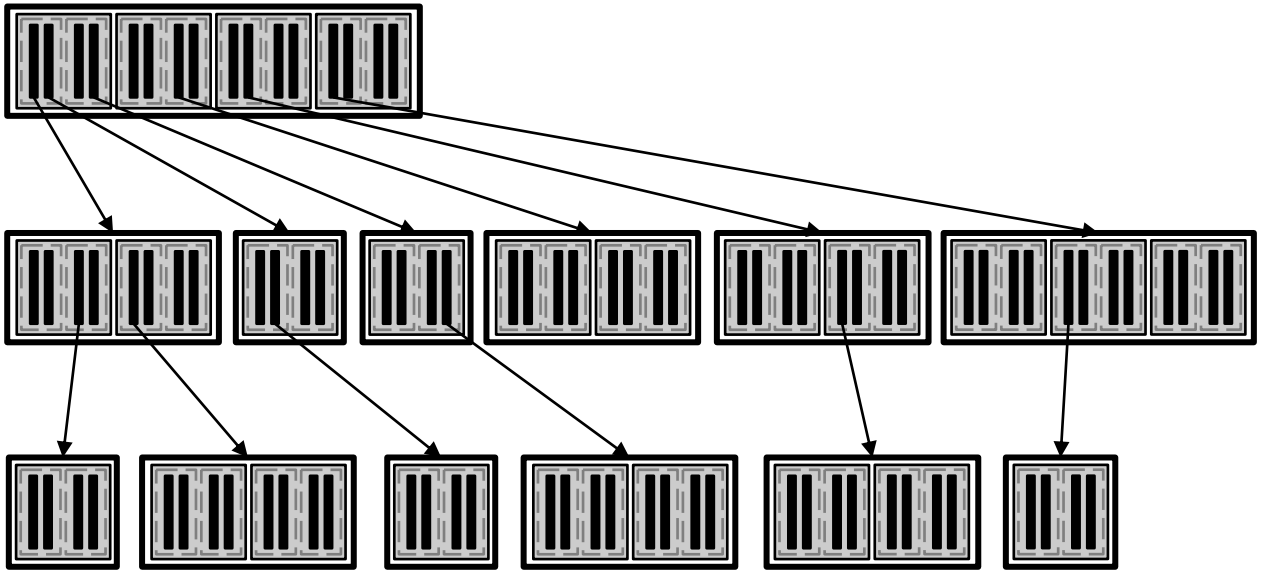
- Launch overhead on the critical path and limited depth of call stack

Motivation: Launch Overhead (I)

Problems:

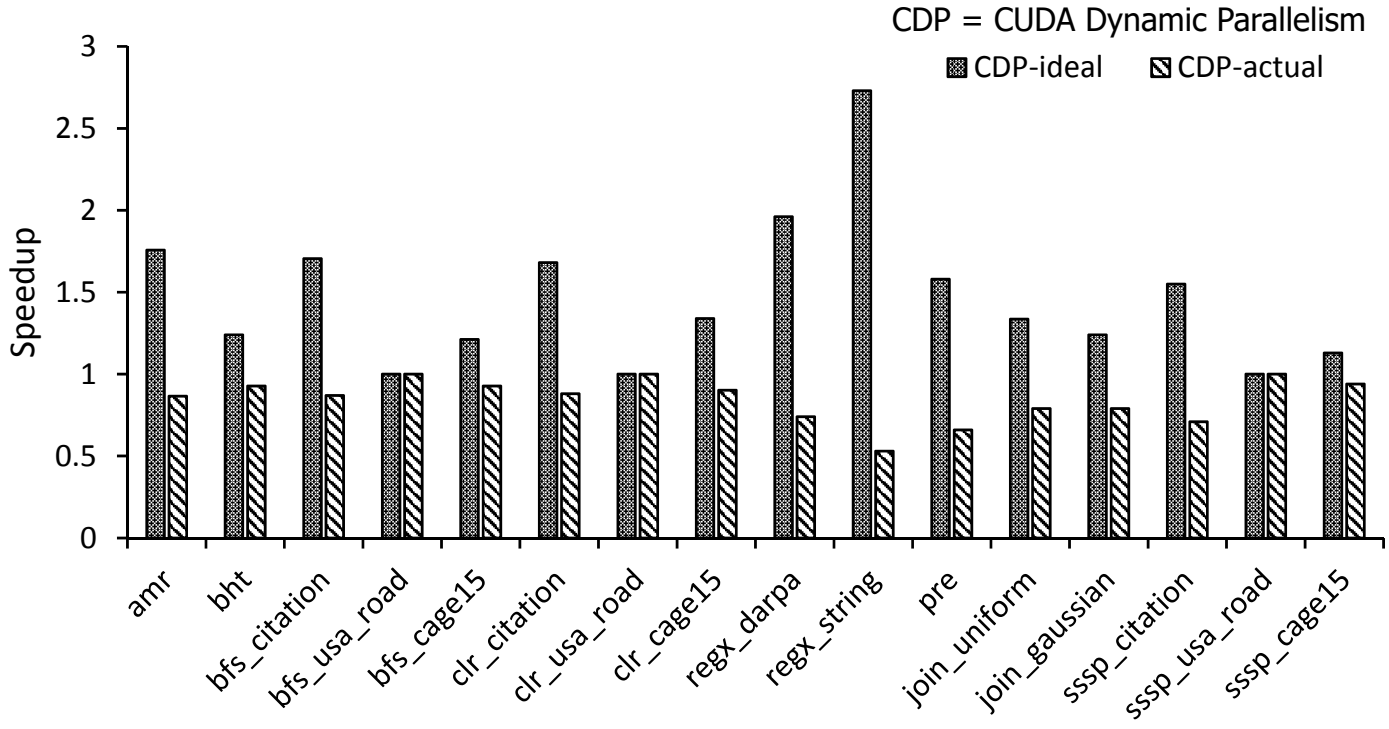
Many kernels incur too much **launch overhead**

Fine-grain kernels **underutilize the GPU resources**



Motivation: Launch Overhead (II)

- Non-trivial overhead of device-side kernel launching
  - Parameter allocation, launch command, dispatch kernel
  - Pending kernels, suspended parents
- Ideal CDP vs. Actual CDP (speedup over non-CDP versions)

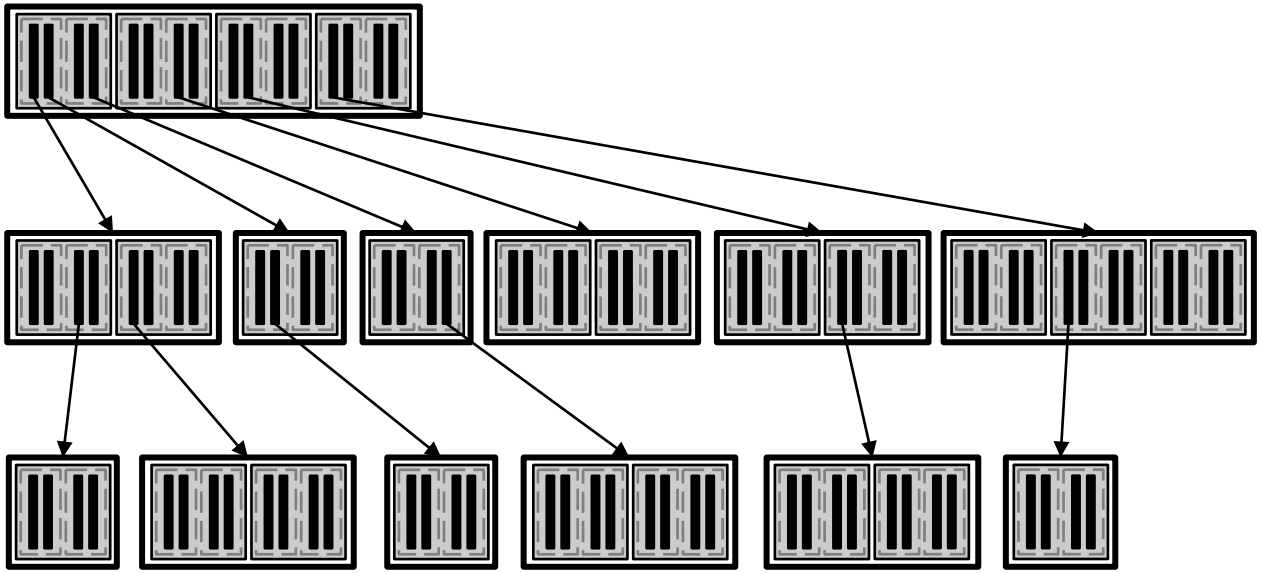


Wang et al., "Characterization and Analysis of Dynamic Parallelism in Unstructured GPU Applications," IISWC 2014

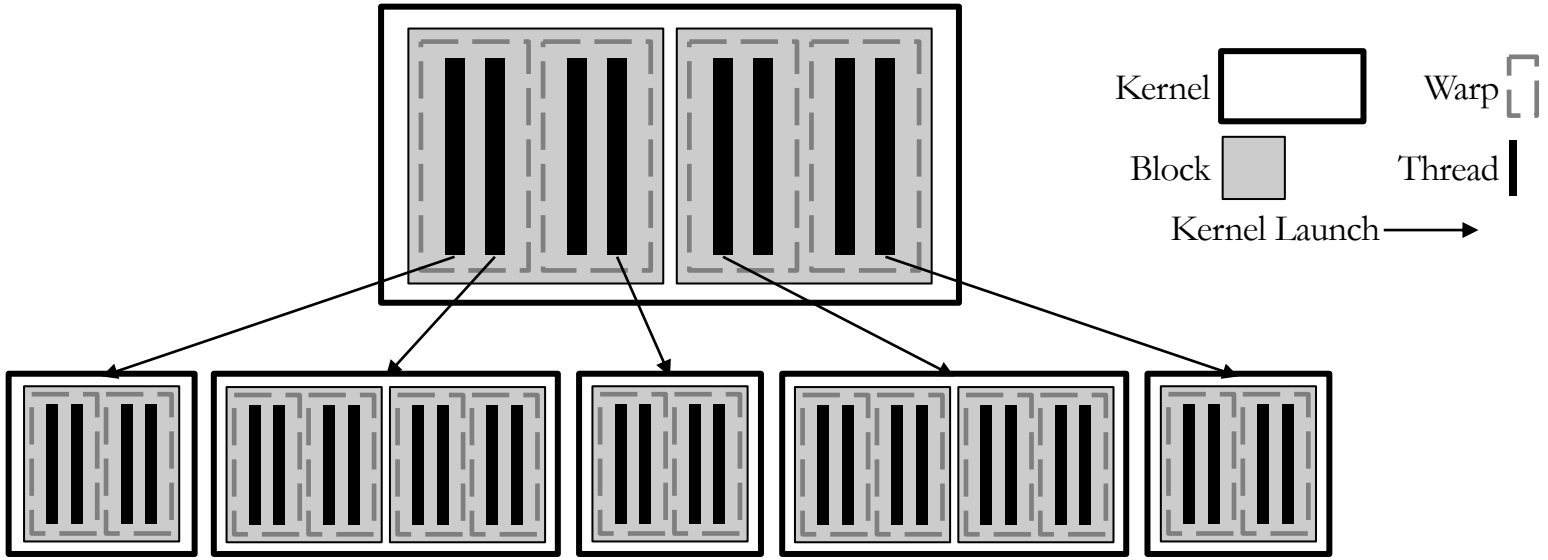
Idea: Kernel Launch Aggregation

Proposed Solution:

Kernel Launch Aggregation

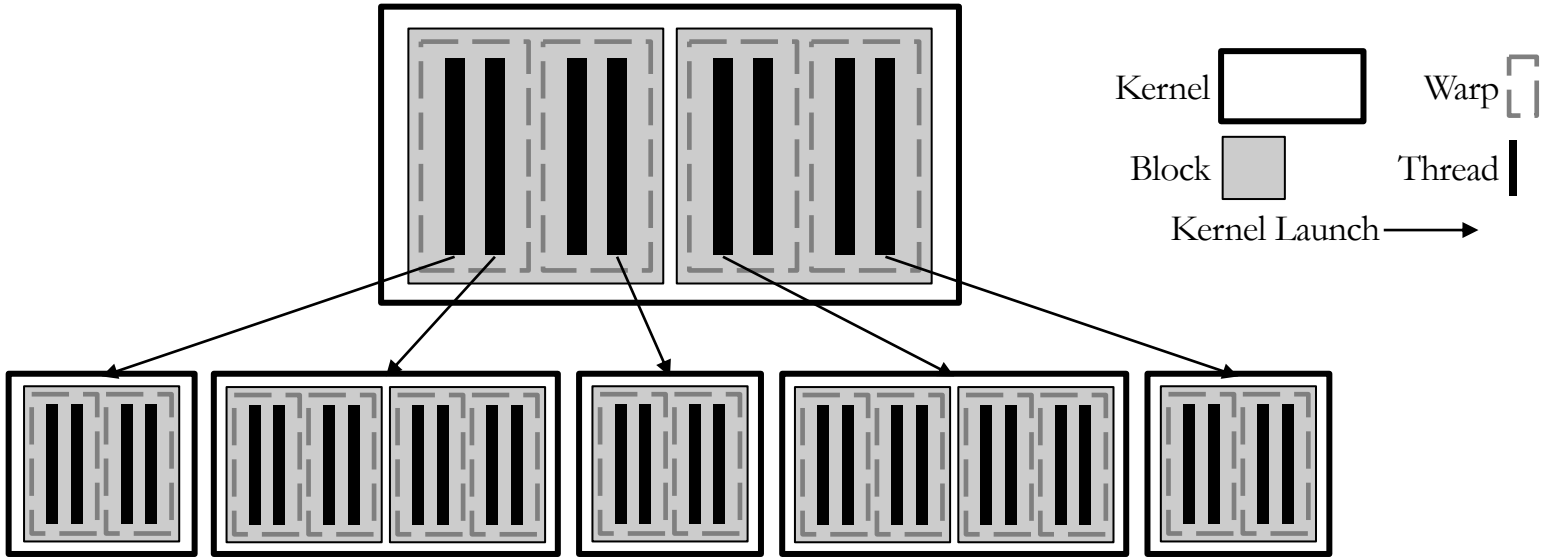


Idea: Kernel Launch Aggregation



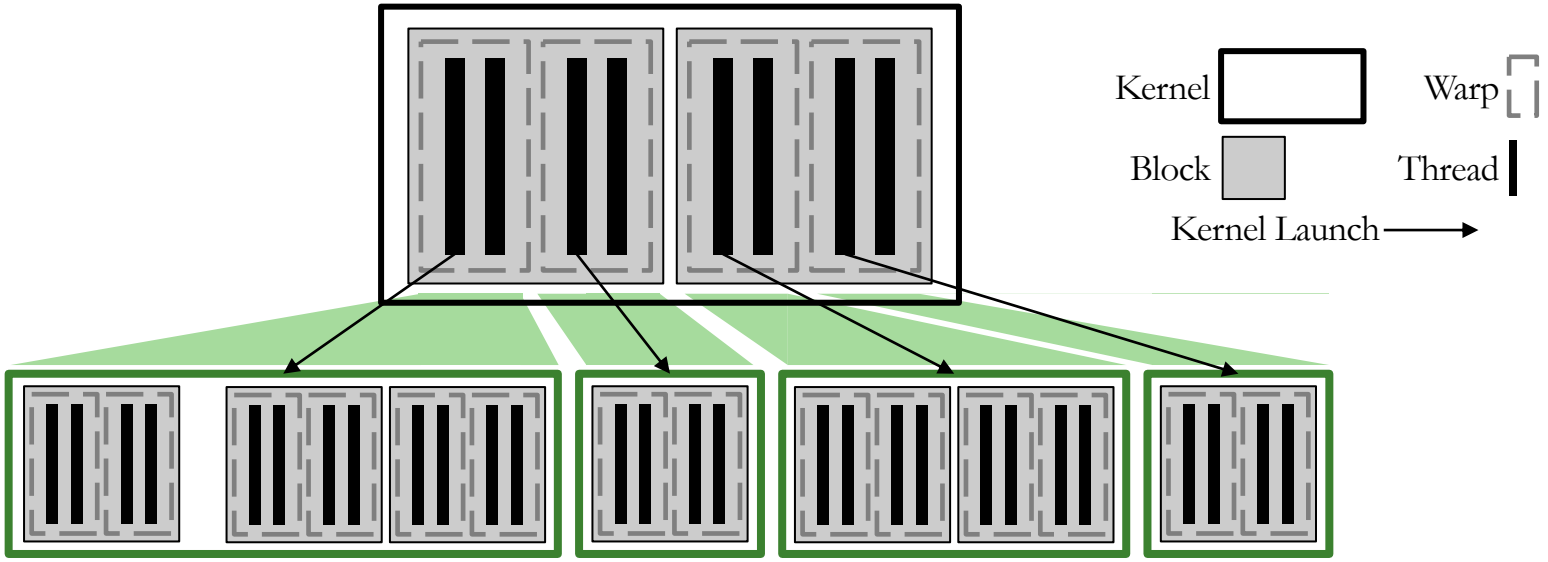
Each thread can launch a separate kernel

Idea: Kernel Launch Aggregation



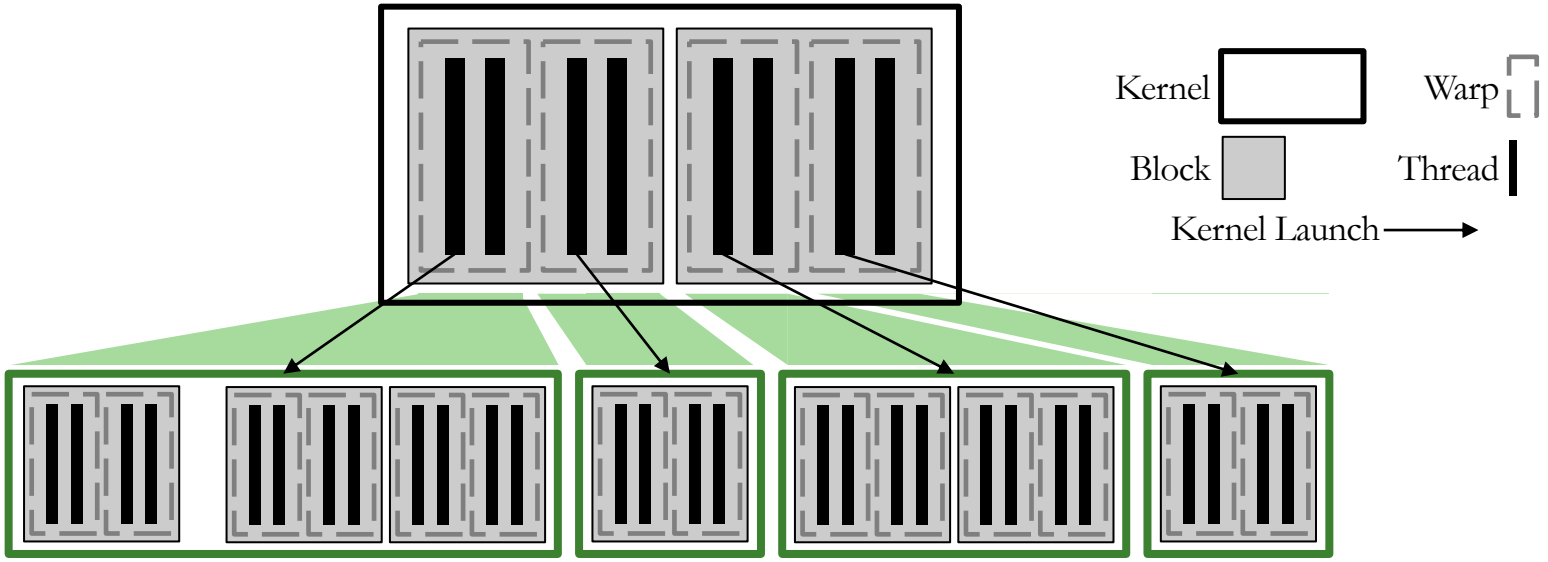
Each thread can launch a separate kernel

Warp-Granularity



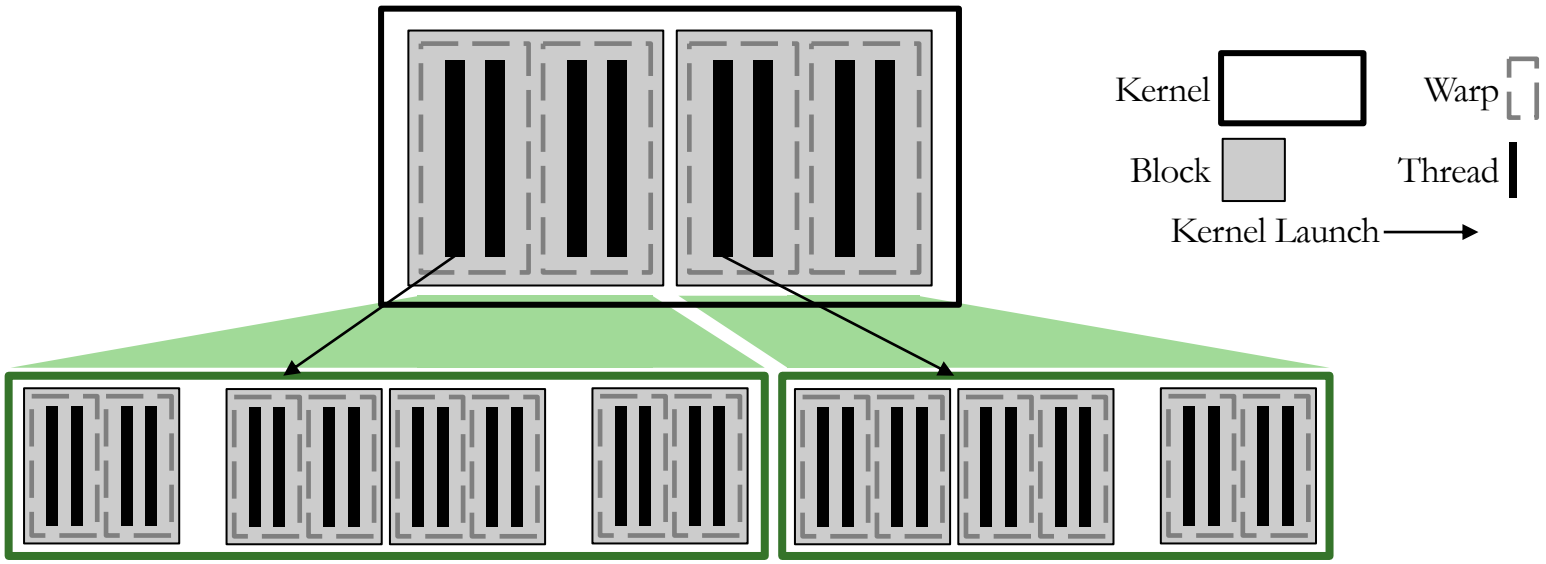
Warp-Granularity Kernel Launch Aggregation

Warp Granularity



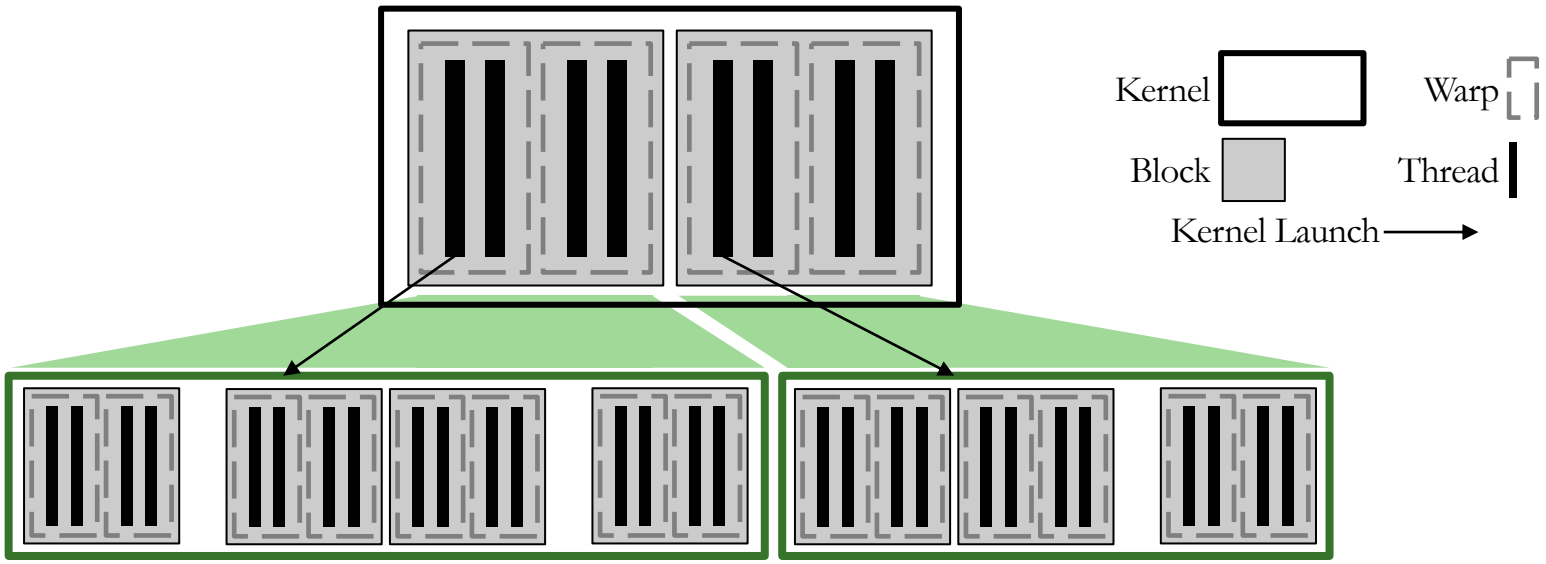
Warp-Granularity Kernel Launch Aggregation

Block Granularity



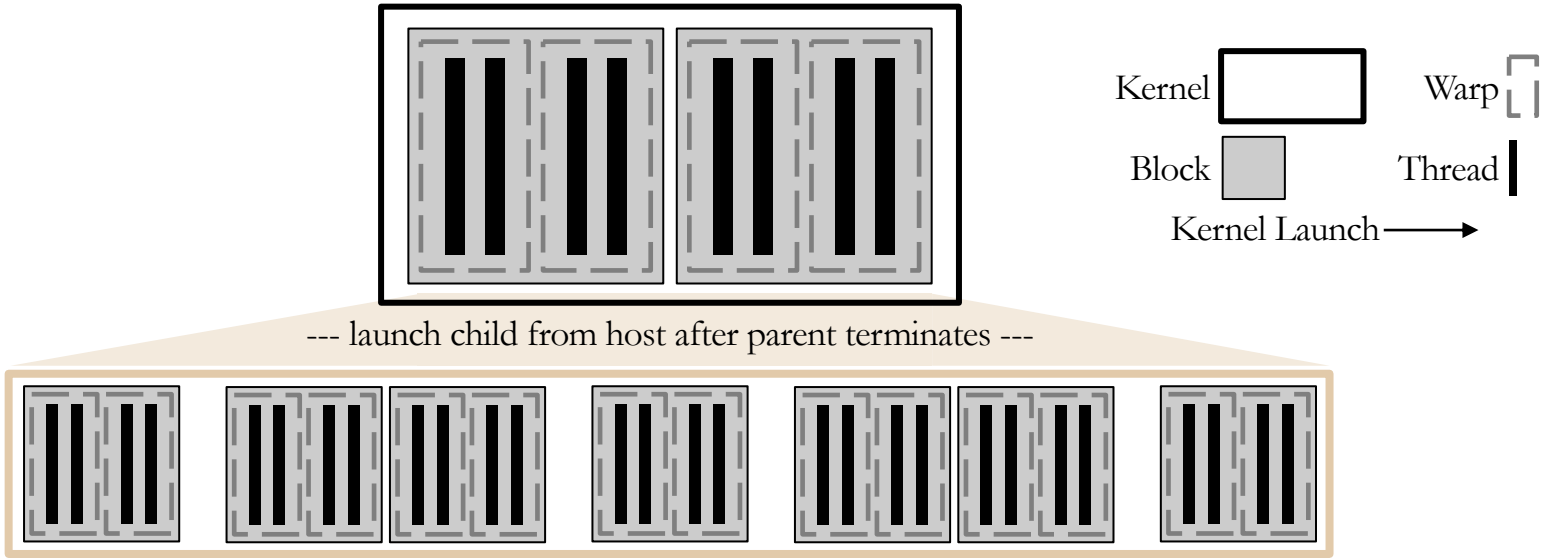
Block-Granularity Kernel Launch Aggregation

Block Granularity



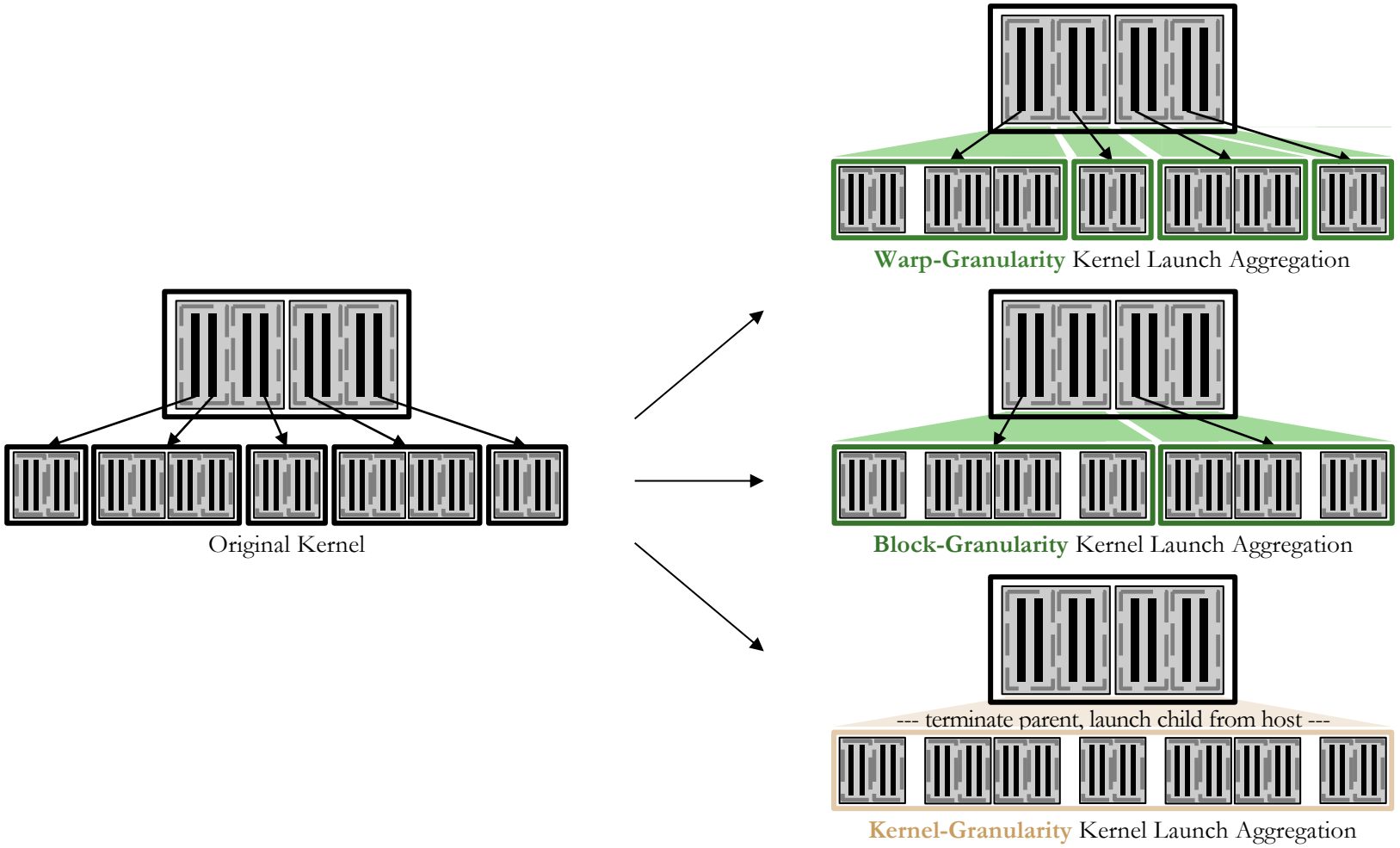
Block-Granularity Kernel Launch Aggregation

Kernel Granularity



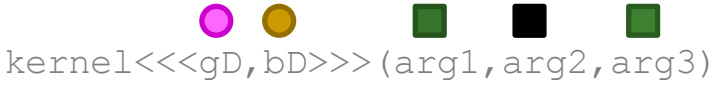
Kernel-Granularity Kernel Launch Aggregation

Kernel Launch Aggregation

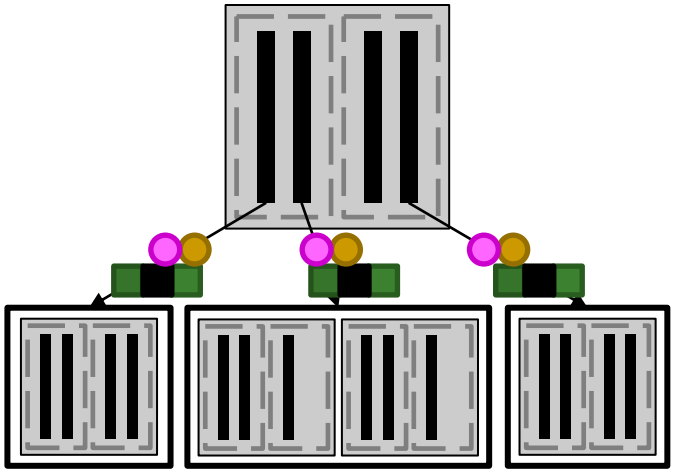


El Hajj et al., "KLAP: Kernel Launch Aggregation and Promotion for Optimizing Dynamic Parallelism," MICRO 2016

Block Granularity Aggregation (I)

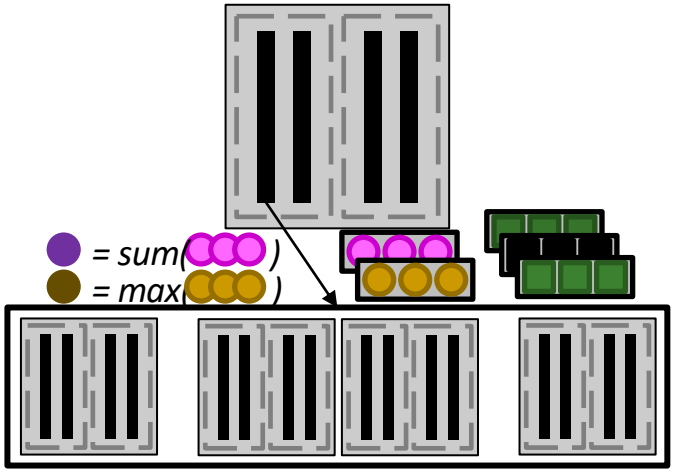


Original Kernel Call



allocate arrays for args, gD, and bD
store `args` *in arg arrays*
store `gD` *in gD array, and* `bD` *in bD array*
new gD = sum of gD array across warp/block
new bD = max of bD array across warp/block
if(threadIdx == launcher thread in warp/block) {
    `kernel_agg<<<`*new gD*`,` *new bD*`>>>`
        `(`*arg arrays, gD array, bD array*`)`
}

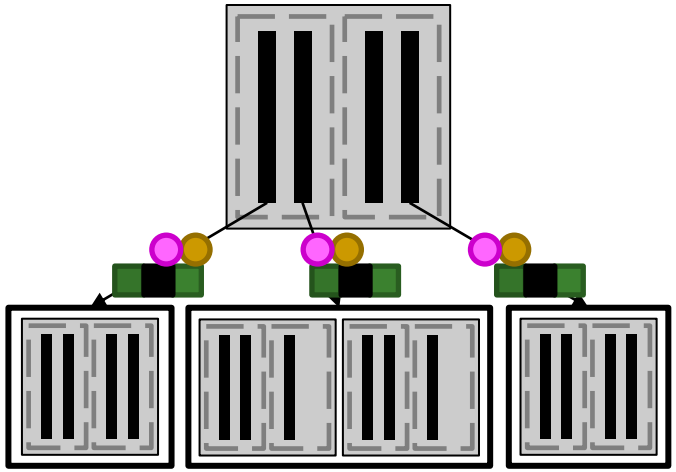
Transformed Kernel Call
(block-granularity aggregation example)



Block Granularity Aggregation (II)

```
__global__ void kernel(params) {
    kernel body
}
```

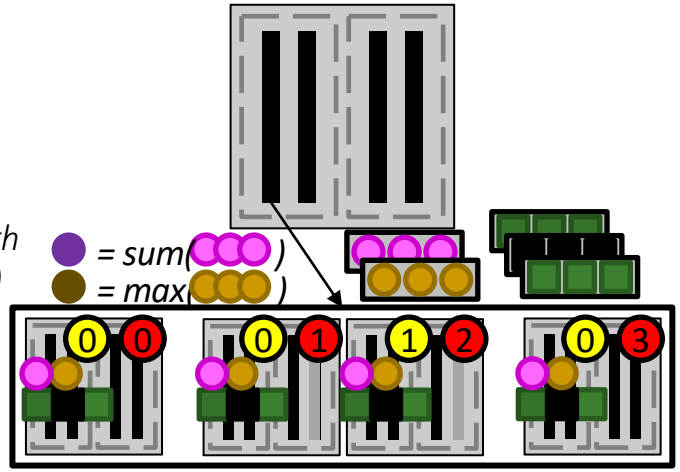
Original Kernel



```
__global__ void kernel_agg(param arrays, gD array, bD array) {
    calculate index of parent thread ( original kernel index) (#)
    load params from param arrays
    load actual gridDim/blockDim from gD/bD arrays
    calculate actual blockIdx (#)
    if(threadIdx < actual blockDim) {
        kernel body   (with kernel launches transformed and with
                       using actual gridDim/blockDim/blockIdx)
    }
}
```

Transformed Kernel

(block-granularity aggregation example)



El Hajj et al., "KLAP: Kernel Launch Aggregation and Promotion for Optimizing Dynamic Parallelism," MICRO 2016

Block Granularity Aggregation (III)

gD =  = { 1, 2, 0, 1 }

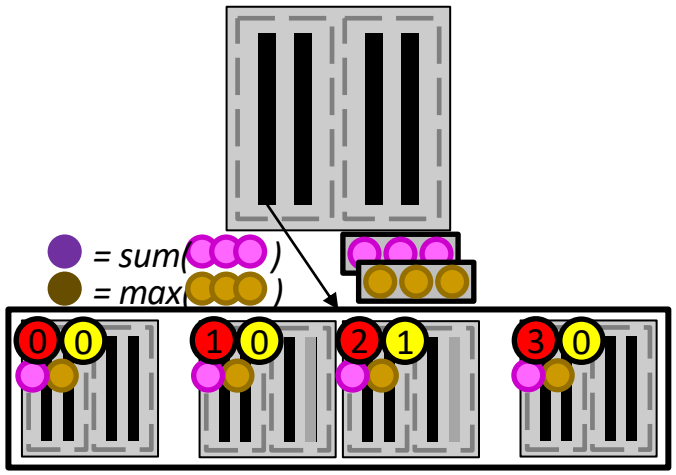
gDs = scan() = { 0, 1, 3, 3, 4 } (performed by parent thread as optimization)

blockIdx.x	=	0	1	2	3
# = *p* | *gDs[p]* ≤ *blockIdx.x* < *gDs[p+1]*	=	0	1	1	3
# = blockIdx.x – gDs[p]	=	0	0	1	0

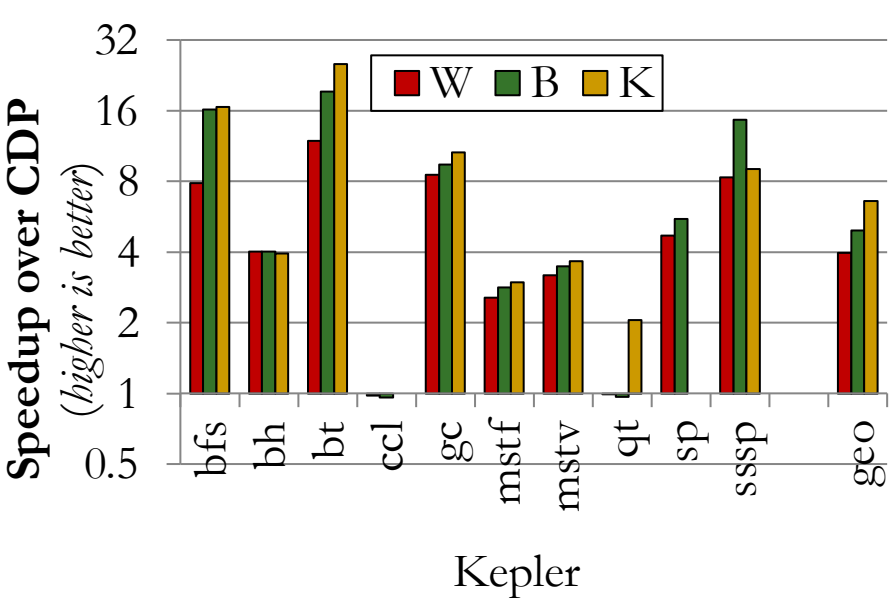
(n-ary search on gDs for *p* satisfying this condition)

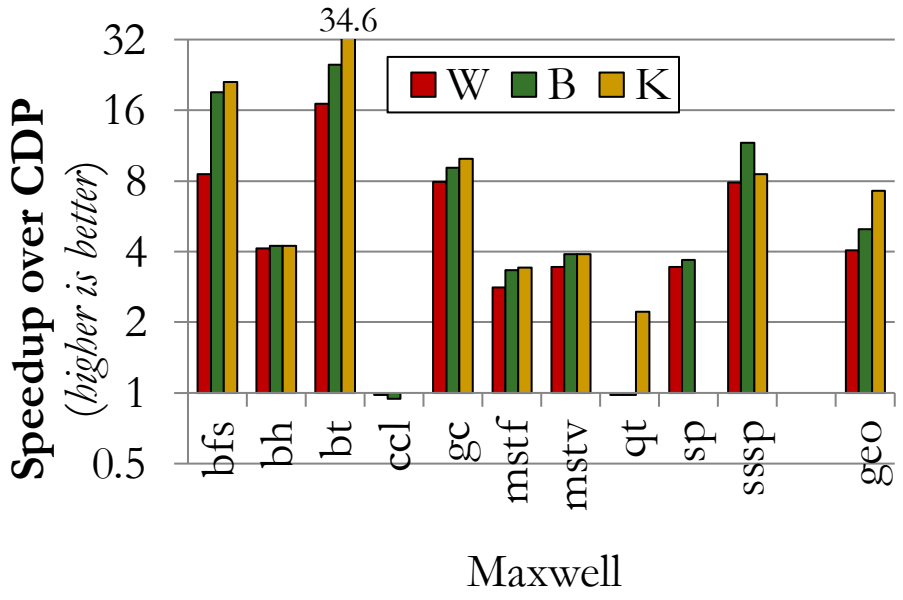
calculate index of parent thread #

calculate actual blockIdx #



Performance of Kernel Launch Aggregation

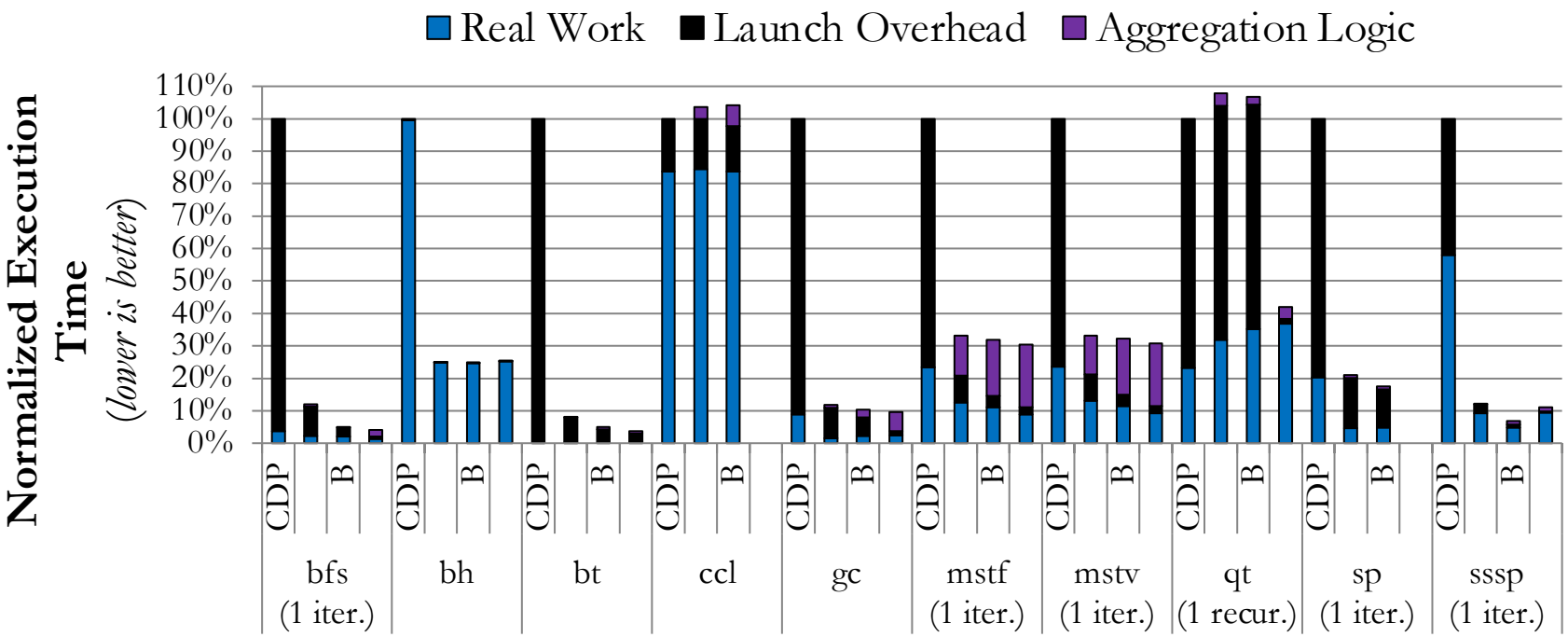




Increasing aggregation granularity improves performance
(geomean speedup of 6.58x for K-aggregation on Kepler)

El Hajj et al., "KLAP: Kernel Launch Aggregation and Promotion for Optimizing Dynamic Parallelism," MICRO 2016

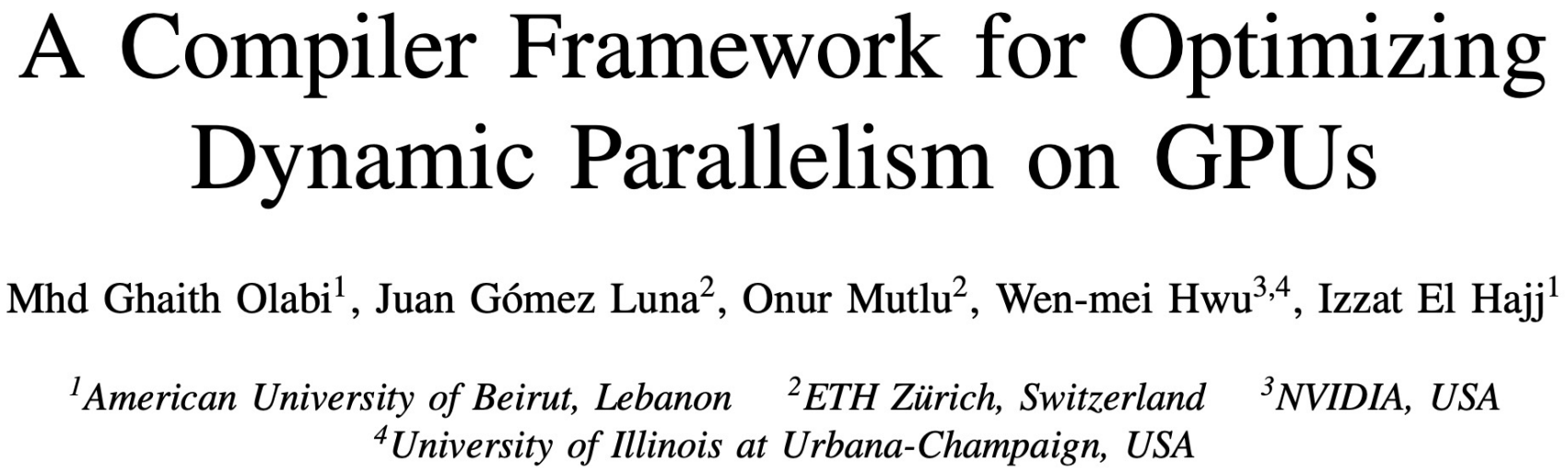
Profiling of Kernel Launch Aggregation



Performance improvement comes from reduced launch overhead and better resource utilization

A Compiler Framework for Dynamic Parallelism (I)

- Mhd Ghaith Olabi, Juan Gomez Luna, Onur Mutlu, Wen-mei Hwu, and Izzat El Hajj,
  "A Compiler Framework for Optimizing Dynamic Parallelism on GPUs"
  *Proceedings of the International Symposium on Code Generation and Optimization (**CGO**)*, Virtual, April 2022.
  [Slides (pptx) (pdf)]
  [Short Talk Slides (pptx) (pdf)]
  [Source Code (Officially Artifact Evaluated with All Badges)]
  Officially artifact evaluated as available, reusable and reproducible.



A Compiler Framework for Optimizing Dynamic Parallelism on GPUs

Mhd Ghaith Olabi[1], Juan Gómez Luna[2], Onur Mutlu[2], Wen-mei Hwu[3,4], Izzat El Hajj[1]

[1]American University of Beirut, Lebanon [2]ETH Zürich, Switzerland [3]NVIDIA, USA
[4]University of Illinois at Urbana-Champaign, USA

A Compiler Framework for Dynamic Parallelism (II)

- Thresholding (as a compiler optimization)
  - A grid is launched only if the number of child threads exceeds a threshold
  - Prior work relies on programmers to apply it manually
- Coarsening of child thread blocks
  - The work of multiple blocks is assigned to a single block
  - Prior work on compiler-based coarsening not specialized for dynamic parallelism
- Aggregation of child grids at multi-block granularity
  - Child grids of multiple blocks are consolidated into a single grid
  - Prior work only compiler-based aggregation only considers warp, block, and grid granularity

- One compiler framework that combines the three optimizations

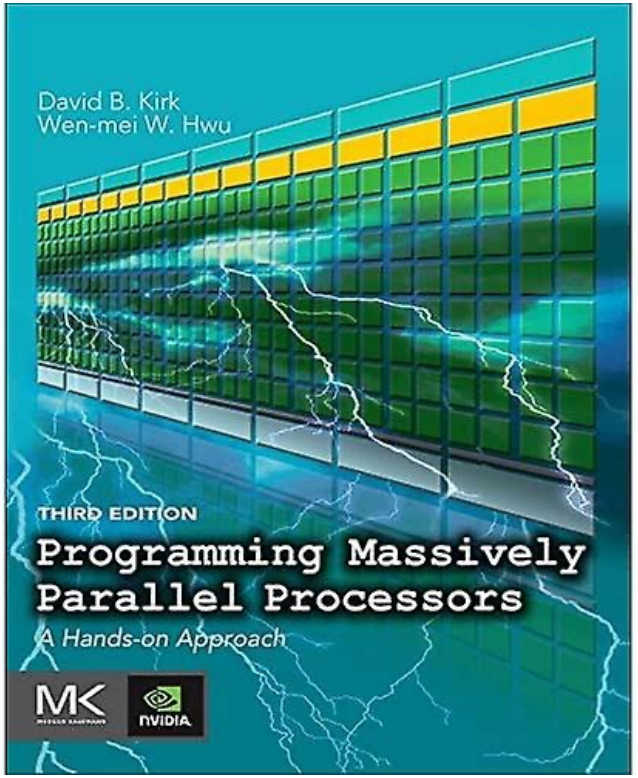
Olabi et al., "A Compiler Framework for Optimizing Dynamic Parallelism on GPUs," CGO 2022

Dynamic Parallelism: Summary

- CUDA Dynamic Parallelism
  - Extends the CUDA programming model to allow kernels to launch kernels
  - Dynamic memory allocation
  - Dynamically discovered work
  - Recursive algorithms
  - Better work balance and more efficient memory usage
- Performance limitations
- Launch overhead
  - Kernel launch aggregation

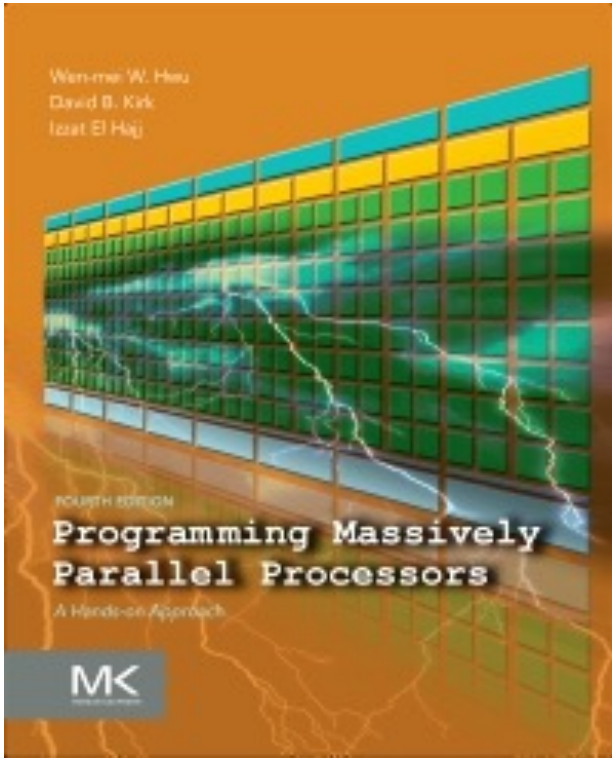
Recommended Readings (I)

- Hwu and Kirk, "Programming Massively Parallel Processors," Third Edition, 2017
  - Chapter 13 - CUDA dynamic parallelism



Recommended Readings (II)

- Hwu and Kirk and El Hajj, "Programming Massively Parallel Processors," Fourth Edition, 2022
  - Chapter 21 - CUDA dynamic parallelism



P&S Heterogeneous Systems

Dynamic Parallelism

Dr. Juan Gómez Luna
Prof. Onur Mutlu
ETH Zürich

Fall 2022

9 January 2023